Université de Toulouse

Université Mohammed V-Agdal de Rabat

THESE EN CO-TUTELLE

Soutenue en vue de l'obtention du titre de

DOCTEUR EN INFORMATIQUE

De l'Université de Toulouse
et de l'Université Mohammed V-Agdal de Rabat

Intégration de la modélisation comportementale dans la conception par points de vue

---

Présentée et soutenue publiquement par

YOUNES LAKHRISSI

Le 02 juillet 2010 à l'Université de Toulouse

Devant le jury composé des membres suivants :

Rapporteurs :

Franck **Barbier**	Professeur à Université de Pau et des Pays de l'Adour
Mourad **Gharbi**	Professeur Habilité à l'Université Mohammed V-Agdal de Rabat
Jean-Pierre **Giraudin**	Professeur à l'UPMF de Grenoble

Examinateurs :

El-Houssine **Bouyakhf**	Professeur à l'Université Mohammed V-Agdal de Rabat
Bernard **Coulette** (Directeur)	Professeur à l'Université de Toulouse II-Le Mirail
Abdelaziz **Kriouile** (Co-directeur)	Professeur à l'ENSIAS de Rabat
Iulian **Ober** (encadrant)	Maître de Conférences à l'Université de Toulouse II-Le Mirail
Mahmoud **Nassar** (co-encadrant)	Professeur Habilité à l'ENSIAS de Rabat

IRIT CNRS - INPT - UPS - UT1 - UTM

- Institut de Recherche en Informatique de Toulouse (IRIT, UMR 5505)
- Laboratoire d'Informatique, Mathématiques appliquées, Intelligence Artificielle et Reconnaissance de Formes de Rabat (LIMIARF).

LIMIARF LABORATOIRE DE RECHERCHE

A mes Parents

A mes Sœurs

A mon Frère

Remerciements

Les travaux présentés dans ce mémoire sont le fruit d'une collaboration franco-marocaine manifestée sous forme d'une cotutelle entre le laboratoire LIMIARF de Rabat et le laboratoire IRIT de Toulouse. Mes remerciements vont au CNRST (Centre National de Recherche Scientifique et Technique) et au réseau STIC Franco-Marocain qui ont assuré le financement de cette thèse, contribuant ainsi à la réalisation de mes travaux de recherche dans des conditions très favorables.

Je remercie sincèrement les Professeurs Frank Barbier, Jean-Pierre Giraudin et Mourad Gharbi, qui ont accepté de juger ce travail et d'en être les rapporteurs. Je les remercie pour l'application avec laquelle ils ont lu mon manuscrit et toutes les questions, riches d'intérêt, qu'ils ont pu soulever et qui sont encore vives dans mon esprit. Je remercie également le professeur El Houssine Bouyakhf d'avoir accepté de participer à l'examen de cette thèse.

Je tiens à remercier vivement mon directeur de thèse Bernard Coulette pour sa disponibilité, ses relectures très minutieuses et son accueil exemplaire ; je n'oublierai jamais ses gestes nobles tels que ses déplacements volontaires pour venir me chercher de ou m'amener à l'aéroport … la liste est longue ! Je remercie également mon co-directeur de thèse Abdelaziz Kriouile pour son aide continue, ses conseils précieux, pour son soutien inconditionnel et son dévouement.

Je tiens également à témoigner ma profonde reconnaissance à mes encadrants Iulian Ober et Mahmoud Nassar. Merci Iulian pour vos conseils constructifs, votre dynamisme et votre efficacité qui ont été un apport considérable dans l'accomplissement de ce travail. Merci Mahmoud pour la relation fraternelle dont vous m'avez octroyé, pour les conseils précieux, pour le soutien permanent, pour m'avoir initié à la recherche et pour les nombreuses discussions techniques que nous avons eues tout au long de ces années.

Mes remerciements vont également à mes amis de l'IRIT et du LIMAIRF. J'adresse mes vifs remerciements aux thésards que j'ai croisés quasi-quotidiennement et avec qui j'ai eu l'occasion et le plaisir de partager des repas et pauses café sympathiques. Je parle particulièrement du trio Adil, Issam et Yaël dont la bonne humeur, la chaleur et l'écoute ont joué un rôle significatif pour l'achèvement de cette thèse. Je parle également de Rhama, Hakim, Samba, Eric et Adel pour leur affection et leur amitié dévouée qui m'ont profondément touché.

Ces cinq années resteront ancrées dans ma mémoire comme une période très enrichissante de ma vie. Cet aboutissement dans mes études ne pouvait se réaliser sans être associé au soutien des personnes les plus proches de moi : ma famille, une valeur à laquelle j'attache énormément d'importance. Je ne saurais jamais assez remercier mes PARENTS, les êtres qui me sont les plus chers. Ils ont eu un rôle essentiel et continu et sans eux aucune réussite n'aurait été possible. Mes sœurs Hajar, Assia et Btissam, admirables, que j'affectionne particulièrement. Mon adorable petit grand frère Zakaria, sa femme Alkhansae, Abderafia et Simohamed ont été mes alliés et leurs encouragements m'ont beaucoup apporté.

Mes remerciements ne peuvent se clôturer sans la mention spéciale à un trio qui compte beaucoup pour moi. Merci Khalid Doumal, Hakim Quartit et Hamza Rayd.

Enfin merci à ceux et celles que je n'ai pas pu citer, mes sincères amitiés et remerciements.

Résumé

La modélisation par points de vue constitue la thématique générale de notre travail de thèse. C'est une approche de modélisation orientée objet, visant l'analyse et la conception des systèmes complexes avec une démarche centrée autour des acteurs interagissant avec le système. Notre équipe a ainsi développé un profil UML appelé VUML (View based UML), qui permet l'élaboration d'un modèle unique partageable à partir de vues associées aux points de vue des acteurs. Cependant, les travaux réalisés sur le profil VUML [Nassar, 05 ; Anwar, 09] ne couvrent pas les aspects comportementaux de la modélisation. En effet, – en proposant la notion de classe multivue – VUML traite les aspects structuraux liés à la composition des vues et au partage des données statiques sans prendre en compte la manière dont ces vues vont réagir, ni comment les synchroniser afin de représenter le comportement des objets multivue (instances d'une classe multivue). Les travaux effectués dans le cadre de cette thèse cherchent à combler ce manque en dotant le profil VUML de nouveaux mécanismes permettant d'exprimer le comportement d'un système. Nous nous sommes concentrés pour cela sur le comportement des objets multivue décrit par des machines à états qui nécessitent des adaptations des concepts de modélisation UML.

Pour réaliser cet objectif nous avons introduit la notion de sonde d'événements, qui permet de spécifier des communications implicites entre les vues par observation d'événements. Ceci permet de découpler des spécifications qui sont a priori fortement interconnectées, de les concevoir séparément par points de vue, selon les préconisations de la méthode VUML, puis de les intégrer sans avoir à les modifier. Nous avons tout d'abord défini le concept de sonde d'événements, identifié les différents types de sondes avec les paramètres associés, puis défini un ensemble de concepts permettant d'enrichir et de manipuler les sondes. Nous avons ensuite proposé une représentation compatible avec VUML des concepts introduits sous forme d'un profil nommé VxUML (extension de VUML). En plus de la définition des éléments du profil (stéréotypes, valeurs marquées, classes de librairie prédéfinies), nous avons développé en OCL des règles de bonne formation (sémantique statique).

Pour illustrer l'intérêt des concepts introduits, nous avons développé une étude de cas en proposant une modélisation par points de vue complète traitant à la fois les aspects structurel et comportemental.

Pour valider concrètement notre approche, nous avons développé, selon une approche IDM, un générateur de code qui prend en entrée une spécification de système en VxUML. Ce générateur utilise les techniques de transformation de modèles liées à la génération de code, et notamment les transformations de modèles indépendants de plate-forme (PIM) vers des modèles spécifiques à une plate-forme (PSM), et des modèles PSM vers le code ; il a été développé dans un premier temps avec le langage cible Java.

Sur le plan méthodologique, nous avons défini une démarche associée à VxUML, qui complète celle de VUML, en permettant d'utiliser d'une manière méthodique les mécanismes dédiés au traitement du comportement. Désormais, le processus de développement VxUML permet une modélisation par points de vue complète, traitant à la fois les aspects structurel et comportemental.

Mots clés : Conception par points de vue, profil VUML, profil VxUML, sonde d'événements, machine à états multivue, composition du comportement.

Abstract

View-based modeling is the main subject of this thesis. It is a variant of the object oriented modeling approach for the analysis and design of complex systems, focusing on the actors that use the system and decomposing the specification according to their needs. With this prospect, our team developed a UML profile named VUML (View based UML), which allows the elaboration of a unique and sharable model accessible according to the view of each of the system's actors. However the achieved work on the VUML profile does not tackle the behavior aspects of the modeling process. The VUML approach address the structural aspects related to the composition of views and to the sharing of data without dealing with the way these views will react, or how to be able to synchronize them in order to obtain the behavior of multiple-view objects (instances of a multi-view class). The achieved work in this thesis aims to fill this gap by providing new mechanism to the VUML profile that allows expressing the behavioral needs of a system. We will focus on describing the individual behavior of multi-view objects by state-machines that require adjustments of UML modeling concepts.

To achieve this objective we introduced the concept of event probes, which allow to specify implicit communication between the view-objects via event observation. This allows decoupling the view specifications that are a priori strongly interconnected, and therefore allows them to be designed separately, in agreement with the VUML method recommendations, and then to be integrated without the need of making modifications. We first define the concept of event monitoring and identified the different types of probes and the appropriate parameters that characterize them. We also define a set of concepts that can extend and manipulate probes. Then we propose a UML-compatible representation of the introduced concepts in form of a profile named VxUML (extension of VUML). In addition to defining the profile elements (stereotypes, tagged value, predefined library classes), we have developed rules for proper formation of the static semantics using OCL to reduce ambiguity in the specification of such concepts. Afterwards, to demonstrate the usefulness of the concepts introduced, we developed a case study through which we sought to provide a full view-based model, addressing both structural and behavioral aspects. At the end, to concretely validate our approach in a model driven engineering setting, we developed a code generator that takes as input a specification developed in VxUML profile. This generator uses the techniques of model transformation related to the code generation, including transformation of platform independent models (PIM) to platform specific models (PSM), and transformation of PSMs to code; the current target language is Java.

Finally, another contribution of this thesis is a process associated with VxUML. It is an approach that complements the approach associated with VUML (which deal only with structural aspects) to be able to use in methodical manner the new mechanisms dedicated to behavioral treatment. Now, the VxUML development process offers a complete view based modeling, that is, dealing with both structural and behavioral aspects.

Keywords: View based modeling, VUML profile, VxUML profile, event observation, multi-view states machine, behavior composition.

Table des Matières

Table des figures

CHAPITRE I. INTRODUCTION GENERALE

Ce travail de thèse a été effectué dans le cadre du réseau STIC franco-marocain en génie logiciel. Il s'est déroulé dans le contexte d'une cotutelle entre l'Université de Toulouse et l'Université Mohammed V-Agdal de Rabat. Les laboratoires d'accueil sont :

- Laboratoire IRIT[1] (Institut de Recherche en Informatique de Toulouse),
  Equipe MACAO (Modèles, Aspects, Composants pour des Architectures à Objets) ;
- Laboratoire LIMIARF[2] de Rabat (Laboratoire d'Informatique, Mathématiques appliquées, Intelligence Artificielle et Reconnaissance de Formes)

I.1. Contexte : modélisation par points de vue et profil VUML

Malgré l'évolution des techniques d'analyse/conception dans le domaine du génie logiciel, la construction de systèmes informatiques reste une tâche extrêmement difficile. En effet, en parallèle de cette évolution, les besoins évoluent également en termes de diversité des acteurs, de demande d'accès personnalisés, de cohérence des données, de sécurité, de persistance des informations, etc. Ces facteurs d'évolution conduisent à l'accroissement de la complexité des systèmes qui se répercute naturellement dans les différentes phases du cycle de vie. Pour être efficiente, la prise en compte de cette complexité doit se faire dès les phases amont du développement (analyse/conception) dans lesquelles se concentrent les activités de modélisation.

Dans le contexte d'un système complexe, la construction d'un modèle global prenant en compte simultanément tous les besoins est souvent impossible. Dans la réalité, soit plusieurs modèles partiels sont développés séparément et coexistent avec les risques d'incohérence associés, soit le modèle global doit être fréquemment remis en cause quand les besoins des acteurs évoluent. La multi-modélisation est généralement l'approche adoptée pour faire face à cette situation. Elle permet en effet une décomposition de la modélisation par facettes, notamment dans la phase de conception où plusieurs modèles peuvent être développés pour représenter des perspectives particulières du système. Les modèles partiels doivent être ensuite composés pour produire le modèle final du système. La multi-modélisation a été déclinée de plusieurs manières dans différents domaines de la modélisation des systèmes ; citons notamment la modélisation par sujets [Harrison et al., 93], la

[1] http://www.irit.fr/

[2] http://www.fsr.ac.ma/LIMIARF/index.htm

modélisation par rôles [Kristensen et al., 96], la modélisation par aspects [Kiczales et al., 97] et la modélisation par points de vue [Kriouile, 95 ; Coulette et al., 96 ; Muller et al., 03].

La modélisation par points de vue constitue la thématique générale de notre travail de thèse. Elle se situe dans la continuité de travaux de notre équipe ayant donné lieu initialement au langage VBOOL [Marcaillou et al., 94] et à la méthode VBOOM [Kriouile, 95]. VBOOM est une approche de modélisation visant l'analyse et la conception par objets de systèmes complexes par une démarche centrée *acteur*. A chaque acteur du système est associé un point de vue. A chaque point de vue s'appliquant sur le système est associé un ensemble de vues. Le résultat d'une telle modélisation est un modèle unique, partageable, accessible suivant plusieurs points de vue.

Pour pallier entre autres la difficulté due au fort non déterminisme dans l'identification des vues avec VBOOM, notre équipe a développé ensuite une autre approche de modélisation par points de vue sous la forme d'un profil UML appelé VUML (*View based UML*) [Nassar, 05]. Dans cette approche, une seule vue est associée au point de vue d'un acteur donné, ce qui facilite grandement la démarche d'analyse sans perdre en richesse d'expression. Le profil VUML introduit un ensemble de concepts dont celui de "classe multivue". Une classe multivue est une entité de modélisation qui permet de décrire l'information en fonction des points de vue des acteurs concernés. Elle est statiquement composée d'une base (partie partagée par les acteurs associés à la classe multivue) et d'un ensemble de vues étendant cette base. Un objet multivue – instance d'une classe multivue – est un objet composite composé d'un objet-base et d'un ensemble d'objets-vue, chacun encapsulant les données et opérations spécifiques à un acteur [Nassar, 05]. VUML introduit également le concept de "composant multivue" [El Asri, 05] qui étend la notion de composant UML2.0.

A ce jour, VUML supporte les aspects structuraux liés à la définition et au partage des données, ainsi qu'à la composition des vues [Anwar, 09]. Mais VUML ne traite pas les aspects comportementaux de la modélisation, en particulier la manière dont les vues actives réagissent à des événements, ni la façon de les synchroniser afin de représenter le comportement des objets multivue. Les travaux effectués dans le cadre de cette thèse cherchent à combler ce manque en dotant le profil VUML de mécanismes permettant d'exprimer les besoins comportementaux d'un système.

I.2. Problématique de spécification du comportement dans VUML

La modélisation comportementale est importante à considérer dans la démarche de conception d'un système complexe, surtout dans le contexte de l'Ingénierie Dirigée par les Modèles (IDM), où l'objectif est d'arriver à une automatisation des phases de post-conception (codage, intégration, validation, etc.). En effet, une telle automatisation requiert un modèle de conception le plus complet possible. La modélisation comportementale en UML peut se faire à plusieurs niveaux d'abstraction, en partant des modèles d'ensemble comme les modèles d'interaction et d'activités qui représentent les interactions et l'enchaînement des activités entre les différents objets ou les composants du système, et en allant jusqu'à la description fine du comportement des objets ou des composants par des machines à états. Les modèles d'ensemble, tels que les diagrammes de séquence, permettent par définition la description d'un comportement selon un point de vue ou la combinaison de plusieurs points de vue.

Dans ce travail de thèse, nous nous concentrons sur la description du comportement individuel des objets multivue en répondant à la question suivante : **est-ce que le comportement des instances d'une classe multivue est lui aussi multivue ?**

Pour traiter cette question, nous avons choisi de spécifier le comportement d'un objet multivue par la notion de machine à états. Pour illustrer la manière d'aborder cette question, nous prenons l'exemple d'une voiture en réparation dans une agence spécialisée. L'état de cette voiture en réparation peut être considéré comme un état multivue, car il est interprété différemment selon chaque type d'acteur (cf. Figure I.1). Ainsi, un mécanicien s'intéresse aux pannes et aux réparations qu'il doit faire, aux outils nécessaires pour effectuer la réparation, aux pièces de rechange. Un responsable atelier voit par contre la réparation du côté logistique, au sens où il s'intéresse aux affectations des pistes de réparation pour effectuer les opérations de maintenance, à la réservation du matériel, aux affectations des pièces de rechange, etc. Pour un client, les détails techniques d'une réparation sont peu importants ; il est plus intéressé par les détails du contrat de réparation, les frais à engager et par la date de fin de la réparation. Les intérêts du chef de l'agence se focalisent quant-à eux sur la rentabilité de la réparation en considérant son coût réel, la durée estimée pour l'achèvement des réparations, et le contrat à établir avec le client.

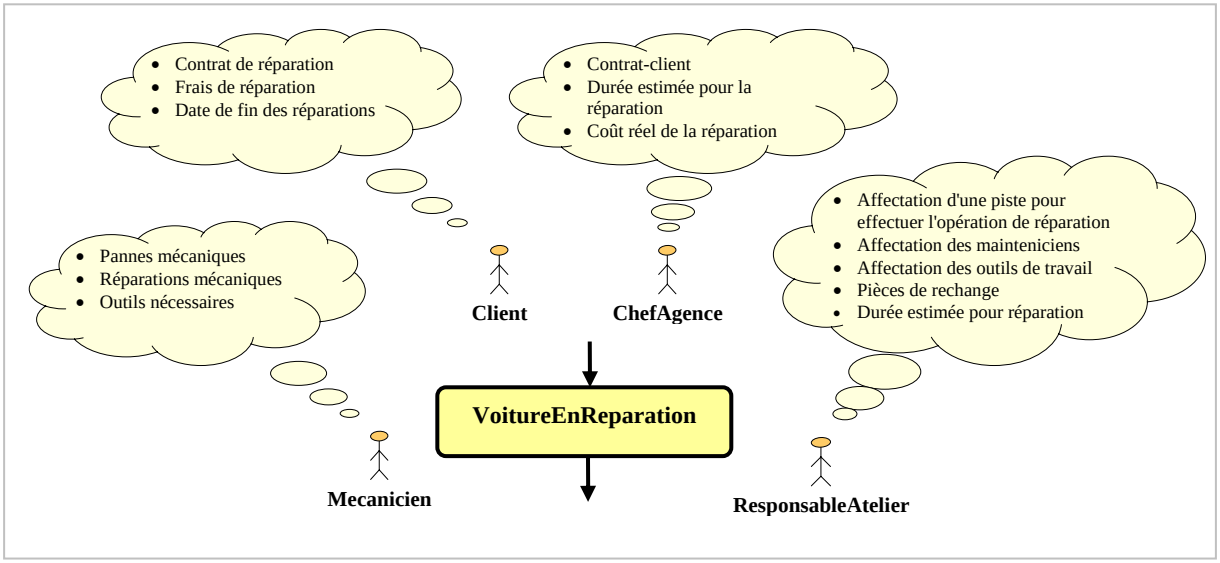


Figure I.1. Points de vue sur l'état d'une voiture en réparation

Sur cet exemple de la voiture, on constate que les besoins liés à l'état « VoitureEnReparation » varient selon les acteurs. Compte tenu des interactions existant entre les acteurs, on voit bien que le comportement d'une instance de classe multivue est lui-même multivue. Cependant, à ce jour, le profil VUML n'intègre pas la manière dont les objets-vue communiquent, ni comment ils collaborent pour réaliser un besoin comportemental donné. En conséquence, les questions suivantes s'imposent de façon pertinente : comment spécifier les comportements individuels des objets-vue ? Comment représenter les interactions entre les objets-vue ? Comment composer les comportements des objets-vue pour représenter le comportement global de l'objet multivue ? Comment garantir la cohérence et l'intégrité du comportement composé ?

I.3. Démarche de résolution et contributions de la thèse

Pour décrire et faire interagir le comportement des vues dans le cadre d'un objet multivue, nous avons tout d'abord exploré les possibilités offertes par UML.

Approche fondée sur les concepts d'UML. Cette approche propose une méthode de description séparée et de coordination des machines à états des objets-vue et de l'objet-base. Le résultat essentiel est l'identification, pour une classe multivue donnée, de deux types de machines à états attachées soit à une classe « base », soit à une classe « vue ». Ces machines à états, qui suivent la spécification UML2.0 [OMG-UML], ont pour rôle de capturer le comportement dynamique des instances de la classe multivue considérée [Lakhrissi et al., 07], [Lakhrissi et al., 08a]. Une machine-vue représente le cycle de vie d'un objet-vue, tandis qu'une machine-base a pour but de créer la cohérence et la coordination entre les machines-vue, et de spécifier le comportement commun aux acteurs. On appelle machine « multivue » l'ensemble formé d'une machine-base et des machines-vue dépendantes.

L'expérience avec cette approche lors d'un passage à l'échelle montre que l'intégration de machines-vue développées séparément requiert de nombreuses modifications et de fastidieuses adaptations de ces modèles quand les vues ont des dépendances importantes entre elles, ce qui arrive fréquemment dans la réalité. Ceci nous a conduit à explorer une deuxième approche permettant de s'affranchir des contraintes liées à l'interdépendance entre les vues. Cette approche consiste à étendre UML par de nouveaux mécanismes dédiés.

Approche par extension des concepts d'UML. Nous avons donc opté pour une autre approche, générique dans son principe et ajoutant dans VUML des mécanismes nouveaux, spécifiques de la modélisation et de la composition des comportements des objets-vue [Ober et al., 08b]. Nous introduisons pour cela la notion de ***sonde d'événements*** pour spécifier des communications implicites entre les objets-vue à travers des observations d'événements. Ceci permet de découpler des spécifications qui sont a priori fortement interconnectées, de les concevoir séparément, puis de les intégrer sans avoir à les modifier. Pour atteindre notre but, nous avons tout d'abord défini le concept de sonde d'événements, identifié les différents types de sondes avec les paramètres associés, puis défini un ensemble de concepts permettant d'enrichir et de manipuler les sondes. Nous avons ensuite proposé une représentation compatible avec VUML des concepts introduits sous forme d'un profil nommé VxUML (extension de VUML). Pour illustrer l'intérêt des concepts introduits, nous avons développé une étude de cas en proposant une modélisation par points de vue complète traitant à la fois les aspects structurel et comportemental. Cette approche a été validée par un prototype permettant de générer du code exécutable à partir d'une modélisation en VxUML.

I.4. Organisation du mémoire

En plus de cette introduction, ce rapport de thèse est organisé en deux parties :

La première partie présente l'état de l'art de notre domaine de recherche ; elle est composée de deux chapitres :

- *Chapitre 2* **- Spécification du comportement dans les langages de modélisation.** Ce chapitre est une introduction à des techniques reconnues de spécification du comportement dans les langages de modélisation objet. Nous décrivons notamment les formalismes intégrés à UML, comme les machines à états, les modèles d'activités, les modèles d'interaction et les modèles d'actions.

- *Chapitre 3* **- Multi-modélisation et VUML.** Dans ce chapitre nous décrivons d'abord des techniques de multi-modélisation proches de notre problématique comme la modélisation par

aspects, la modélisation par rôles et la modélisation par sujets. Nous donnons ensuite un aperçu sur les approches de composition du comportement, notamment celles à base d'aspects qui sont basées sur des principes similaires à ceux de notre proposition. Nous consacrons le reste de ce chapitre à la description du contexte applicatif de la thèse qui est le langage VUML ainsi que la démarche d'analyse/conception associée. Nous discutons des limites actuelles de VUML et nous introduisons la problématique de la spécification comportementale par points de vue.

La deuxième partie est consacrée à notre proposition de traitement de la spécification et de la composition du comportement dans VUML. Elle est composée de trois chapitres :

- ***Chapitre 4* - Extension de VUML pour la modélisation du comportement.** Ce chapitre est divisé en deux sections principales. Dans la première section, nous présentons notre première approche de modélisation utilisant les concepts existant dans UML. A la fin de la section, nous discutons les limites de cette solution qui ont motivé le développement d'une autre approche basée sur la notion de sonde d'événements. La deuxième section, dédiée à cette approche, définit le concept de sonde d'événements et les concepts associés. Nous proposons une représentation compatible avec UML des différents concepts introduits et définissons une extension du profil VUML permettant la spécification comportementale des objets multivue. En plus de la définition des éléments du profil (stéréotypes, *tagged values*, classes de librairie prédéfinies), cette section décrit également les règles de sémantique statique (cohérence de modèles) formalisées en OCL.

- ***Chapitre 5* - Démarche VxUML et application à un cas d'étude**. Nous proposons tout d'abord une démarche de développement associée à VxUML, qui constitue un guide méthodologique sur l'utilisation des concepts introduits dans cette thèse. L'utilité des concepts introduits dans le chapitre 4 est montrée ensuite à travers une étude de cas : système de gestion d'une agence de réparation de voitures.

- ***Chapitre 6* - Prototype VxUML.** La notion d'observation est un mécanisme de spécification du comportement puissant, mais qui nécessite une implémentation efficace pour être utile en pratique. Pour prouver la faisabilité de ces concepts, une mise en œuvre sous la forme d'un outil de prototypage rapide (génération de code Java) est présentée. Cet outil a été développé dans un cadre IDM avec des techniques de transformation de modèles et en utilisant le langage ATL (*Atlas Transformation Language*).

Dans la conclusion de ce rapport (***chapitre 7***), nous donnons dans un premier temps une synthèse des travaux effectués dans le cadre de cette thèse. Nous mettons ensuite en évidence les points originaux de notre travail. Enfin, nous énonçons les perspectives possibles à ce travail de recherche, tant sur le plan théorique que sur le plan de l'outillage et des applications.

Après la liste des références bibliographiques, nous fournissons trois annexes. L'annexe A présente le méta-modèle PivotVxUML développé pour représenter les modèles pivot servant de base pour les opérations de manipulation des modèles VxUML au sein de notre prototype. L'annexe B présente les règles de vérification sémantique des éléments manipulés dans le prototype. L'annexe C présente un extrait du code Java généré par notre prototype.

CHAPITRE II. SPECIFICATION DU COMPORTEMENT DANS LES LANGAGES DE MODELISATION

Sommaire

II.1. Introduction

D'une manière générale, on peut résumer les objectifs de cette thèse par les deux points suivants : (1) comment spécifier le comportement dans les modèles-vue (modèles partiels issus d'une modélisation selon les points de vue des acteurs du système) ; et (2) comment composer les comportements individuels des vues pour obtenir le comportement complet des objets multivue (instances des classes multivue). Pour traiter ces deux points, nous avons réalisé un état de l'art sur les différentes techniques utilisées actuellement pour spécifier le comportement (le présent chapitre), et sur les techniques adoptées pour la composition des comportements (le chapitre suivant).

Ce chapitre est structuré en deux parties. Nous présentons dans la première partie les langages référence dans le domaine de spécification du comportement. Nous passons en revue des exemples représentatifs de ces langages, notamment les langages de description du comportement à base de scénarios et à base d'états. A la fin de cette première partie, nous présentons une synthèse comparative de ces deux types de langage. Dans la deuxième partie, nous décrivons les formalismes intégrés à UML pour la description du comportement comme les modèles d'interaction et d'activités et les modèles à base de machines à états. Nous concluons ce chapitre par une synthèse globale.

II.2. Langages de spécification du comportement

Modéliser un système revient à déterminer sa structure statique et son comportement en se basant sur les trois perspectives standard suivantes :

Modélisation de la structure statique : cette perspective déclare les familles d'objets et les relations qui les relient. Elle montre les éléments structurels comme : les classes, les associations, les interfaces, les attributs, etc.

Modélisation des comportements inter-objets : cette vision d'ensemble est utile dans la spécification des besoins dans les phases initiales d'analyse-conception. Elle montre les interactions entre les objets potentiels du système. Elle donne des visions partielles du système global sous forme de scénarios. Par leur nature partielle et intuitive, les scénarios sont les plus adaptés pour exprimer les exigences.

Modélisation du comportement intra-objet : cette perspective s'intéresse à la représentation du cycle de vie commun aux objets d'une même classe. Elle décrit le comportement complet d'un objet en montrant les différents états et transitions possibles à l'exécution.

Cette section est composée de deux sous-sections. Dans la première, nous passons en revue des exemples de langages référence dans la description des comportements à base de scénarios. Dans la deuxième sous-section, nous décrivons des exemples de langages référence dans la description du comportement intra-objet à base d'états. Nous terminons cette section par une synthèse sur les différents langages étudiés.

II.2.1. Langages de comportement inter-objets à base de scénario

Dans les premières phases du processus de développement, les fonctionnalités des systèmes sont intuitivement définies par des exigences informelles et par des descriptions graphiques. Par leur nature partielle et intuitive, les scénarios sont les plus adaptés pour exprimer les exigences. L'objectif des approches à base de scénarios à un double but : (i) donner des visions partielles du système global et (ii) montrer les séquences d'événements et les interactions entre les objets potentiels du système. Il est important de noter qu'une spécification basée scénario décrit les modalités de communication entre objets, et non pas les algorithmes et les calculs sur des données.

La plupart des langages à base de scénario adoptent un ensemble de notions communes, parmi lesquelles on peut citer :

Ligne de vie : représente un élément participant dans une interaction entre objets. Il peut être un acteur humain ou un objet logiciel. Les diagrammes d'interaction se focalisent sur la communication entre les lignes de vie. Cette communication peut prendre différentes formes : appel de méthodes, envoi de signaux, création/destruction d'objets, etc. Toutes ces formes de communication sont désignées sous le terme de *message*.

Message : c'est un élément de communication unidirectionnelle entre objets qui déclenche une activité dans l'objet destinataire. On distingue deux types de messages, (i) les messages synchrones, pour lesquels l'émetteur se bloque en attente d'une réponse, et (ii) les messages asynchrones, pour lesquels l'émetteur continue son évolution indépendamment de la réponse du récepteur.

De nombreux langages ont été développés pour la description des modèles basés sur les scénarios. Amyot et Eberlein [Amyot et al., 03] fournissent un aperçu étendu de quinze notations de scénarios. Dans cette section, nous présentons des exemples de ces langages à base de scénario représentatifs du domaine. Ces langages sont : le langage MSC (*Message Sequence Charts*) [ITU-MSC, 00], le langage LSC (*Live Sequence Chart*) [Damm et al., 01], et le langage UCM [ITU-UCM, 02]. Concernant les diagrammes d'interaction d'UML, ils sont présentés dans la section II.3.3.1.

II.2.1.1. Langage MSC (Message Sequence Chart)

Le diagramme de séquence de messages, ou MSC, est un formalisme de description de scénarios. C'est un langage de spécification graphique normalisé par l'ITU (*International Telecommunication Union*) en 1993 avec la recommandation Z.120, où est définie la syntaxe du langage, mais pas sa sémantique. Ce formalisme permet d'exprimer des séquences finies d'interactions asynchrones entre entités communicantes. Les diagrammes MSC ont un domaine d'application étendu. Le domaine de référence d'application des diagrammes MSC est la spécification du comportement communicationnel des systèmes temps réel, particulièrement les systèmes de télécommunication. Les diagrammes MSC peuvent être utilisés pour spécifier des prescriptions, des interfaces, des simulations, des validations, des cas de test et des documentations pour ce type de systèmes.

Cependant, malgré son large usage, le langage MSC présente plusieurs limites et a subi par conséquent plusieurs extensions. La principale limitation des MSC est leur faible puissance expressive. En effet, la sémantique du langage MSC définit essentiellement un ordre partiel sur les événements, basé sur l'ordre causal entre l'envoi et la réception de messages et entre les événements initiés par un

même objet. Les événements et les actions des instances n'ont par contre aucune interprétation : seul compte l'ordre dans lequel les événements sont produits. Dans ces conditions, un MSC est incapable de spécifier précisément le comportement réel du système.

Pour surmonter ces lacunes, l'ITU a étendu la syntaxe des MSC en 1996 puis en 2000 en ajoutant des éléments de modélisation de données ainsi que des opérateurs de composition (séquentielle et parallèle), afin de pouvoir exprimer des séquences infinies d'événements. La sémantique de ce nouveau langage a été donnée quelques années plus tard et est intégrée à la recommandation Z.120 [ITU-MSC, 00]. Le langage MSC a inspiré également une partie du standard UML2, où les diagrammes de séquence reprennent la plupart des constructions de MSC-2000.

Une autre extension importante enrichissant le langage MSC a été définie par Damm et Harel en 2001 [Damm et al., 01] sous le nom de LSC (*Live Sequence Chart*). Nous présentons dans la section suivante les principaux ajouts du langage LSC par rapport au langage MSC.

II.2.1.2. Langage LSC (Life Sequence Chart)

Le langage LSC [Damm et al., 01], propose une notation avec une sémantique formelle, plus expressive que les MSC. Avec la notion de "température", les LSC font la distinction entre les scénarios existentiels qui *peuvent* apparaître dans un système (scénarios qualifiés par une température froide ou *cold*) et les scénarios universels qui *doivent* apparaître dans un système (qualifiés par une température chaude ou *hot*). Cette notion de température est supportée et exprimée par la plupart des éléments constituant ce langage et peut être présentée graphiquement. Par exemple, dans une interaction donnée, les messages qualifiés de *cold* sont des messages qui *peuvent* être reçus, alors que les messages (*hot*) *doivent* être reçus sinon l'interaction est interrompue. Une condition peut être *cold*, signifiant qu'elle *peut* être vraie (sinon le contrôle est transmis à l'extérieur du bloc courant) ou *hot* signifiant qu'elle *doit* être vraie (sinon le système s'interrompt). De la même façon, la progression suivant les axes temporels peut être *hot*, exprimée par un axe continu forçant la progression, ou *cold*, exprimée par un axe en pointillé dans lequel la progression n'est pas obligatoire. Le langage LSC a apporté un ensemble de nouveaux concepts que nous résumons dans les points suivants [Brill et al., 04] :

Liveness et températures : comme mentionné ci-dessus, la température peut être chaude ou froide. Une température chaude indique que la progression est imposée, sinon elle est facultative. Graphiquement, les températures chaudes sont représentées par des lignes pleines, et les températures froides par des lignes en pointillé. Dans la Figure II.1-a, la portion entre l'envoi du message Sync2 et la réception du message de retour Ret_Syn2 est froide et dessinée en pointillé.

Conditions : comme dans le langage MSC, une condition est représentée par un hexagone. Elle a deux variantes : *obligatoire* et *possible*. Une condition obligatoire doit être satisfaite, la violation de cette condition est considérée comme une erreur. Une condition *possible* qui est invalidée conduit simplement à la sortie du diagramme qui l'englobe. Les conditions obligatoires sont dessinées par des lignes pleines (le cas de Cond1 dans la Figure II.1-a) et les conditions possibles par des lignes en pointillé.

Co-régions : dans un diagramme LSC, l'enchaînement par défaut des éléments se définit de haut en bas le long de l'axe de vie des instances. L'ordre entre les instances est induit seulement par l'échange de messages et par les conditions qui s'appliquent sur plusieurs instances. Une corégion

signifie qu'il n y a aucun ordre imposé aux événements qu'elle contient, c.-à-d. que les événements peuvent se produire dans n'importe quel ordre. Une corégion dans LSC garde la même signification qu'une corégion dans MSC mais avec la différence qu'elle permet aux événements de se produire simultanément. Graphiquement, une corégion est représentée par une ligne pointillée parallèle à la ligne de vie de l'instance.

Quantification : la quantification représente la distinction entre le comportement obligatoire et le comportement possible au niveau des diagrammes. On distingue deux modes : le mode existentiel (exprime la possibilité) et le mode universel (exprime l'obligation). Le mode universel n'a pas d'équivalent dans le langage MSC, il indique que le comportement présenté dans le LSC doit être accompli par toutes les exécutions. Graphiquement, la quantification est décrite par le style de bordure du diagramme : une frontière dessinée par des lignes pleines indique un diagramme universel, et celle dessinée par des lignes en pointillé indique un diagramme existentiel (c'est le cas du diagramme de la Figure II.1-a).

Activation et Pré-chart : dans les LSC, l'activation d'un diagramme peut être réalisée par trois moyens (cf. Figure II.1-b) : (i) par un événement *e*, (ii) par une condition *c* appelée condition d'activation (AC), ou (iii) par un diagramme dit "pre-chart". On recourt à la troisième solution dans le cas où une simple condition d'activation ou un événement sont insuffisants pour exprimer la pré-condition d'un besoin.

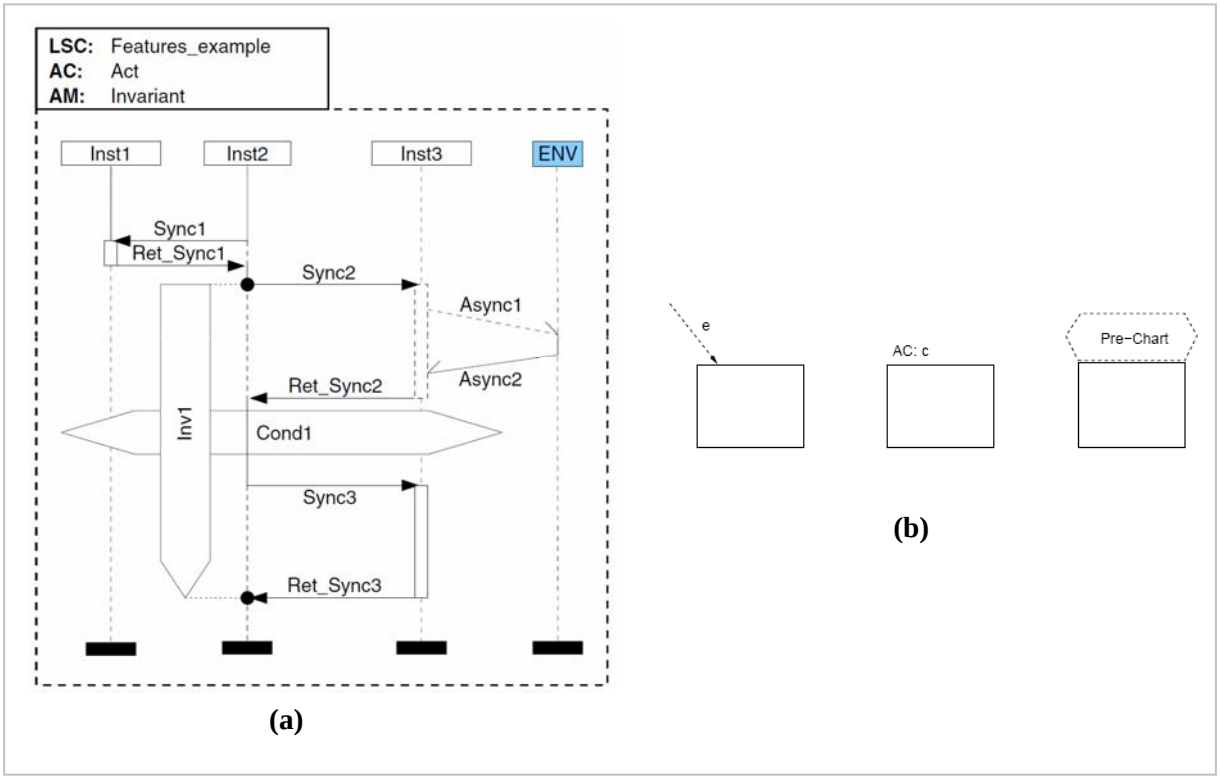


Figure II.1. Aperçu de la notation LSC [Brill et al., 04]

Avec ces éléments de spécification détaillée du comportement et sa sémantique formelle, les LSC sont aussi expressifs que les langage à base d'états, et deviennent donc un moyen intéressant pour les outils de spécification et d'analyse de cas d'utilisation [Harel et al., 05a]. Concernant les outils support de LSC, on trouve essentiellement "play-engine" [Harel et al., 05b]. Cet outil permet de créer

et manipuler les diagrammes LSC ; il implémente également des mécanismes pour synthétiser des machines à états des objets participant dans un système à partir des diagrammes LSC.

II.2.1.3. Langage UCM (Use Case Maps)

Le langage UCM est un langage de spécification et de composition de scénarios. Il représente les aspects comportementaux à un niveau d'abstraction élevé, et constitue un guide aux développeurs dans la phase de mise en œuvre des fonctionnalités du système. Le point fort des diagrammes UCM est leur capacité d'intégrer plusieurs scénarios (dans un diagramme semblable à une carte), et donc de pouvoir mieux traiter l'architecture et le comportement d'un ensemble de scénarios. Les *Use Case Maps* sont apparus en 1995 et sont devenus une recommandation de l'ITU-T en 2002 [ITU-UCM, 02].

Le langage UCM se présente comme une notation visuelle permettant de décrire les besoins fonctionnels et de conception pour les systèmes réactifs et répartis. Il décrit les scénarios sous forme de relations entre les différentes responsabilités. Une responsabilité est un terme qui signifie une opération, action, tâche, fonction, etc., et qui peut se situer le long du déroulement d'un scénario. Afin d'arriver à des réalisations concrètes, la spécification UCM peut être raffinée en utilisant d'autres modèles beaucoup plus détaillés, tels que le langage MSC (cf. section II.2.1.1) ou les diagrammes de séquence UML (cf. section II.3.3.1). Un diagramme UCM de base est composé au moins des trois éléments suivants (cf. Figure II.2) :

- **Start-point :** l'exécution d'un scénario commence à un *start-point*. Un *start-point* est symbolisé par un cercle rempli représentant les pré-conditions et les événements déclencheurs.
- **Responsabilités :** Les responsabilités sont des activités abstraites qui peuvent être raffinées. Elles sont décrites en termes de fonctions, tâches, procédures ou événements. Les responsabilités sont représentées par des croix et sont rattachées à des composants.
- **End-point :** le chemin d'exécution se termine en un *end-point*. Les *end-points* sont symbolisés par des barres représentant les post-conditions et les effets résultants.

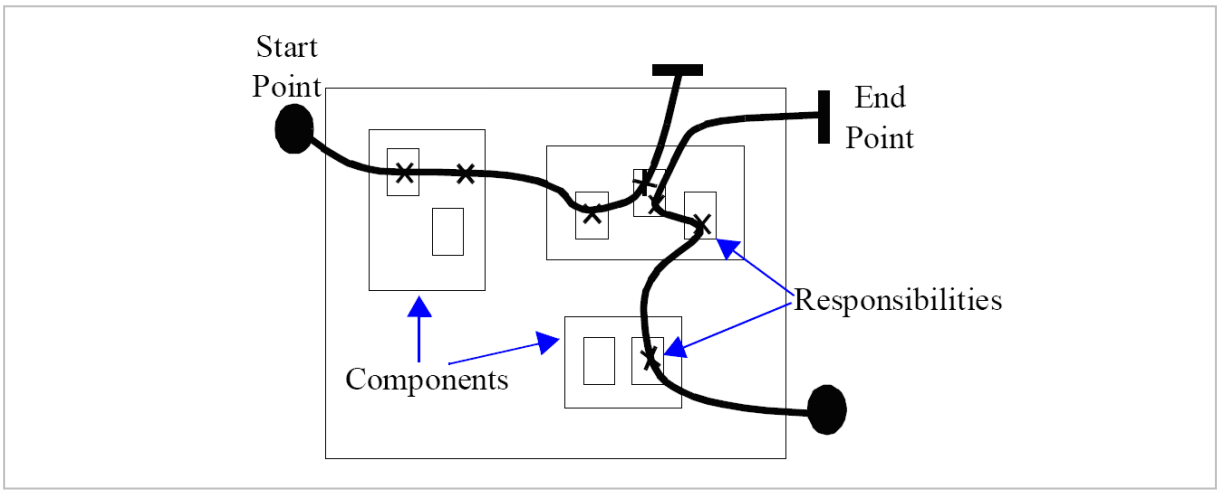


Figure II.2. Aperçu de la notation UCM [UCM-Tutorial, 99]

Quand un UCM devient trop complexe, il est possible de le structurer en sous-diagrammes UCM appelés plug-ins. Ainsi, on peut cacher les détails des plug-ins et les représenter d'une manière iconifiée (sous forme de diamant ou *stub*). Les différents scénarios et plug-ins d'un diagramme UCM peuvent être structurés en séquence comme en parallèle en utilisant les opérateurs OR-fork, OR-join, AND-fork et AND-join respectivement (cf. Figure II.3). OR-fork, représente le cas où un scénario est divisé en deux chemins alternatifs ou plus. OR-join, capture la fusion de deux scénarios indépendants ou plus. AND-fork, découpe un control en deux contrôles concurrents ou plus. AND-join, capture la synchronisation de deux scénarios concurrents ou plus [UCM-Tutorial, 99].

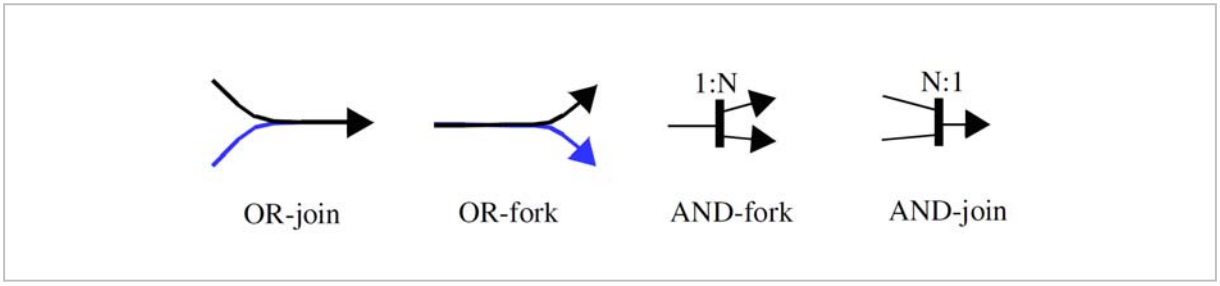


Figure II.3. Les opérateurs OR et AND utilisés dans le langage UCM [UCM-Tutorial, 99]

Concernant les outils support des UCM, on trouve "*UCM Navigator* (UCMNAV)" [UCM-Tool]. Cet outil permet de créer et manipuler les UCM, d'effectuer la vérification syntaxique, et de générer les descriptions équivalentes en XML.

II.2.2. Langages de comportement intra-objet à base d'états

Les langages de comportement à base d'états-transitions décrivent le comportement interne d'un objet à l'aide d'un automate à états finis. Ils spécifient habituellement le comportement d'une instance de classe ou de composant, mais parfois aussi le comportement interne d'autres éléments tels que les sous-systèmes. Un diagramme d'états-transitions offre une vision complète et non ambiguë de l'ensemble des comportements possibles de l'élément auquel le diagramme est attaché. Par contraste, un diagramme d'interaction n'offre qu'une vue partielle correspondant à un scénario sans spécifier comment les différents scénarii interagissent entre eux. A l'inverse, lorsque le comportement décrit est intra-objet, la vision globale du système n'apparaît pas sur ce type de diagramme puisqu'ils ne s'intéressent qu'à un seul élément du système à la fois, indépendamment de son environnement.

Ce type de langage s'intéresse à la représentation du cycle de vie commun aux éléments d'une même nature. Il décrit le comportement complet d'un composant du système (objet/sous-système) en montrant les différents états et transitions possibles à l'exécution. Les états modélisent un ensemble de conditions spécifiques satisfaites à des moments donnés de la vie du composant, et qui déterminent son comportement à ces moments. La plupart des langages à base d'états adoptent un ensemble de notions et de concepts communs, parmi lesquels on peut citer :

- **Classes réactives et classes passives** : Dans un système orienté objet, on distingue deux types de classes : les classes dites *réactives* dont le comportement nécessite le développement d'un diagramme d'états-transitions, et les classes dites *passives* qui sont des classes statiques de

données ne présentant pas de comportement interactif avec les entités du système. Le diagramme d'états-transitions associé à une classe réactive spécifie comment cette dernière traite les messages qu'elle reçoit, ainsi que les réactions qu'elle produit sous forme d'actions.

- **Etat** : Un état représente une situation durant la vie d'un objet pendant laquelle : (i) il attend un certain événement ; (ii) il satisfait une certaine condition ; (iii) il exécute une certaine activité. En plus de la succession d'états normaux correspondant au cycle de vie d'un objet, le diagramme d'états comprend également de pseudo-états, dans lesquels l'objet n'a vocation à passer que de manière transitoire. Par exemple, l'état initial du diagramme d'états correspond à la création de l'instance, l'état final du diagramme d'états correspond à la fin de vie (destruction) de l'instance, un état-condition correspond à un choix qui est fait à un instant donné au cours de l'exécution, etc.
- **Transition** : Une transition décrit la réaction d'un objet lorsqu'un événement se produit. En général, une transition possède un événement déclencheur, une condition de garde qui doit être satisfaite pour que la transition soit activée, un effet (action) produit par l'activation de la transition, et un état cible.
- **Evénement** : Un événement spécifie qu'il s'est passé quelque chose de significatif, localisé dans le temps et dans l'espace. Dans le contexte des machines à états finis, il représente l'occurrence d'un stimulus qui peut déclencher une transition entre états.

Dans la section suivante, nous donnons deux exemples de langages représentatifs dans le domaine de modélisation du comportement basés sur la notion d'états/transitions : les statecharts de D. Harel [Harel, 87] et le fragment du langage SDL (*Specification and Description Language*) de l'ITU-T [ITU-SDL, 00] utilisé pour décrire le comportement des processus SDL. Concernant le diagramme d'états d'UML [OMG-UML], il est présenté plut tard dans la section II.3.3.3.

II.2.2.1. Statecharts

Le langage *Statecharts* a été défini en 1984 par D. Harel [Harel, 84 ; Harel, 87], et a été rapidement adopté comme notation visuelle pour la conception des systèmes concurrents et réactifs. Il est basé sur le concept principal de machine à états finis, et inclut également d'autres concepts tels que les états hiérarchiques et concurrents pour décomposer et modéliser des comportements beaucoup plus complexes. Dans Statecharts, il n'y a pas de notion de composant et de décomposition structurelle d'un système ; en revanche, le comportement de tout un système est décrit par une composition hiérarchique de machines à états. La sémantique de cette composition est décrite d'une manière très précise et implémentée par l'outil Statemate [Harel et al., 96]. Elle repose sur la diffusion (*broadcast*) de messages entre les composants parallèles et sur une interprétation synchrone du parallélisme.

Plusieurs variétés de statecharts ont été identifiées dans la littérature, et chacune a sa propre sémantique. Van der Beek donne un aperçu de 20 variétés de statecharts dans son travail [Beek, 94]. Dans le reste de cette section nous présentons la variante Rhapsody qui est un exemple référence de statecharts.

Rhapsody [Rhapsody-Guide, 04] a été parmi les premiers outils à fournir une sémantique rigoureuse et exécutable pour les statecharts orientés objet. Plusieurs de ses principes fondamentaux ont été adoptés par le standard UML et intégrés dans sa spécification au niveau du diagramme d'états.

Dans Rhapsody, le comportement d'un système réactif est défini par l'ensemble des exécutions possibles. Une exécution est une série d'étapes instantanées dont chacune représente un état dans le cycle de vie du système. Comme montré dans la Figure II.4-a, une étape peut se composer de micro-étapes (*microsteps*). Un système, dans un état initial, répondant à un déclencheur, peut subir une série de *microsteps*, et cela jusqu'à ce qu'il atteigne un état final. Dans ce statut il est prêt pour un éventuel déclenchement.

Il existe trois types d'états : les états-ou (*or-state*), les états-et (*and-states*) et les états simples (*basic states*). Un *or-state* est constitué de sous-états reliés entre eux par l'opérateur «OU exclusif», un *and-state* est constitué par des états orthogonaux rassemblés par l'opérateur «AND», alors que les états simples sont les plus bas dans la hiérarchie d'états, et n'ont pas de sous-états. Dans la Figure II.4-b, les états S, B, C, D sont des *or-states*, alors que A est un *and-state,* tandis que B1, B2, C1, C2, D1, D2, E sont des états simples. Dans Rhapsody, un état racine (*root-state*), le plus haut dans la hiérarchie, est créé implicitement. Dans notre exemple, le *root-state* a l'état S comme sous-état.

La configuration active d'un système désigne l'ensemble des états des objets à un moment donné. Elle comprend : le *root-state*, exactement un seul sous-état pour chaque *or-state*, et tous les sous-états composant un *and-state*. Un exemple de configuration active du statechart de la Figure II.4 est : {B1, B, C1, C, D2, D, A, S, racine}.

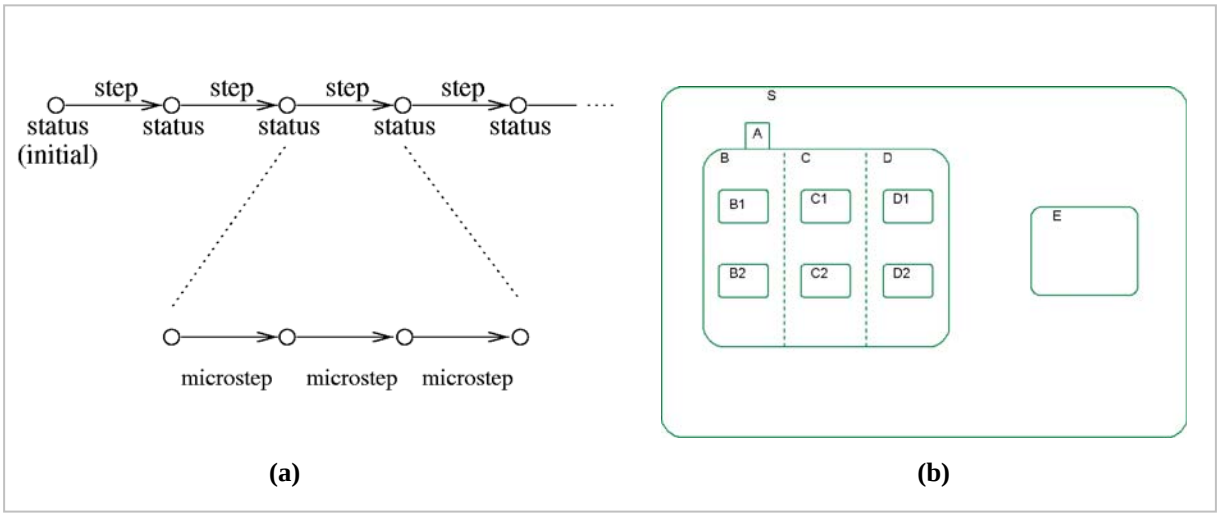


Figure II.4. (a) Progression du système par étapes, (b) Hiérarchie des états dans Rhapsody

Dans Rhapsody, le statechart communique avec les entités du système par l'intermédiaire de deux mécanismes : la communication asynchrone en utilisant les événements, et la communication synchrone en utilisant des opérations de déclenchement (*triggered operation*s). Chaque classe définit l'ensemble des événements qu'elle peut recevoir. La raison principale motivant l'utilisation des événements est que l'objet expéditeur peut continuer son travail sans attendre la réaction du récepteur envers cet événement. Les opérations de déclenchement sont des services fournis par une classe afin de mener des communications synchrones avec ses clients. Puisque son activation est synchrone, une opération de déclenchement peut renvoyer une valeur de retour au client. À la différence des événements, les opérations de déclenchement font partie de la définition de la classe, elles ne sont pas des entités indépendantes. La raison justifiant l'intégration de ce type d'opérations dans Rhapsody est de permettre l'utilisation des statecharts dans les architectures ne supportant pas les événements.

II.2.2.2. Langage SDL (Specification and Description Language)

SDL est un langage de modélisation formel destiné à la spécification et à la description des systèmes réactifs, principalement de télécommunication. Ce langage est une recommandation (sous la référence Z.100) créée et maintenue par l'ITU-T [ITU-SDL, 00]. La première version de cette recommandation a été lancée en 1974, puis elle a connue plusieurs évolutions ; la version utilisée actuellement date de 2007.

Le langage SDL fournit des concepts pour la description et la structuration des systèmes au niveau statique (données) et au niveau comportemental. La description des données est fondée sur les types de données, d'objets et de valeurs. La description du comportement est fondée sur les machines à états finis communiquant par messages.

Le langage SDL décrit le comportement d'un système sous forme d'une séquence de déclencheur/réaction (ou bien, événement/effet). Il fait appel à des concepts structurels qui facilitent la spécification des grands systèmes et des systèmes complexes. Il convient généralement à la description des systèmes réactifs. Ses applications couvrent le cycle de développement entier, qui va de la description des exigences jusqu'à l'implémentation. C'est un langage particulièrement riche qui peut être utilisé à la fois pour des spécifications informelles, formelles ou partiellement formelles. L'utilisateur doit choisir les parties appropriées du langage SDL en fonction du niveau de communication souhaité et de l'environnement dans lequel le langage sera utilisé. Un modèle SDL peut être employé dans différentes phases de développement :

- Analyse des exigences : dans cette phase, SDL permet de construire des modèles abstraits capables de capturer les fonctionnalités du système. Une description informelle est également permise.
- Conception : les modèles de conception SDL permettent de raffiner la spécification des fonctionnalités, ils permettent d'ajouter des détails à l'architecture du système et aux différentes fonctionnalités.
- Implémentation : SDL fournit un langage de programmation impératif qui supporte l'ajout de librairies, le parallélisme, la communication native, la description du comportement sous forme de déclencheur/réaction, etc.
- Validation : pour la phase de validation, le langage SDL utilise deux méthodes, la vérification et le test. La vérification est une validation formelle des propriétés du système en se basant sur des définitions mathématiques [SDL-Annexe-F1, 00]. Le test vérifie si le système réagit comme prévu pour un échantillon d'exécutions du système. Le test est dirigé par le modèle SDL du système en question.
- Documentation : le langage SDL permet une représentation graphique facile et lisible, qui peut être employée pour documenter l'architecture ainsi que le fonctionnement d'un système.

La sémantique formelle du langage SDL est composée de deux parties : (i) la sémantique statique qui comprend la grammaire, les conditions de validité structurelle, et les règles de transformation [SDL-Annexe-F2, 00], et (ii) la sémantique dynamique [SDL-Annexe-F3 00].

Concernant les outils support de SDL, on trouve ObjectGéode [ObjectGeode-site], RTDS de PragmaDev [PragmaDev-site] et Tau de Télélogic [Tau-site].

II.2.3. Synthèse

Les modèles basés sur les scénarios et les modèles basés sur les états fournissent deux facettes différentes des systèmes réactifs. Les scénarios apportent une aide aux concepteurs dans l'extraction et la représentation des fonctionnalités des systèmes. Ils sont naturels à comprendre et faciles à communiquer. Cependant, avec les modèles basés sur les états, le code peut être produit automatiquement et donc les concepteurs de logiciel trouvent que ces modèles sont plus près de l'exécution. Ces deux facettes ne sont pas indépendantes, mais au contraire fortement reliées. En effet, les scénarios représentent des vues partielles sur les fonctionnalités du système ; chacune de ces fonctionnalités est traitée par un objet ou une collaboration d'objets, révélant ainsi des rôles et des états potentiels dans le cycle de vie des objets en question. On peut dire que les scénarios participent au développement du cycle de vie des objets du système étudié. Si le passage entre ces deux types de modèles comportementaux est possible, il conduira à l'augmentation de l'efficacité du procédé global du développement logiciel.

Plusieurs travaux de recherche ont visé à créer une passerelle entre ces deux types de modèles. Les travaux présentés dans [Amyot et al., 03] donnent une synthèse des différentes approches proposées au cours de la dernière décade. La Figure II.5 suivante montre les différents types de transformations possibles à partir des modèles à base de scénarios vers des modèles à base de machines à états.

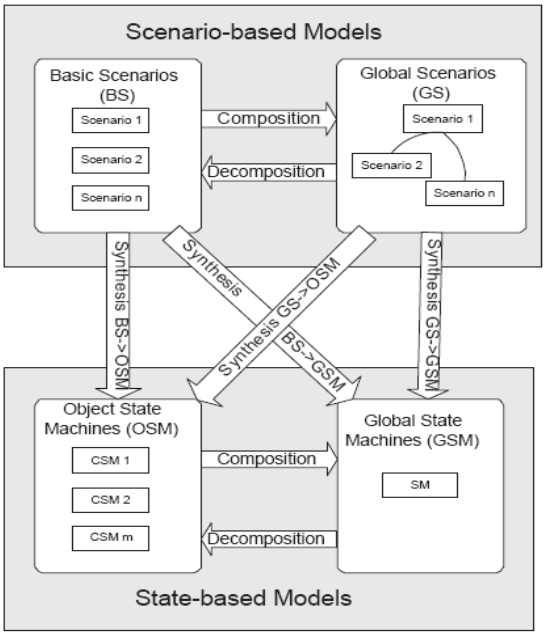


Figure II.5. Différents types de transformation scénarios/machines à états [Liang et al., 06]

À titre d'exemple, on peut citer l'approche Play-In/Play-Out [Harel et al., 00-b] qui permet de synthétiser, à partir des spécifications inter-objets en LSC, des machines à états-transitions des objets participant dans ces diagrammes LSC. On peut citer également l'approche de Castejon [Castejon, 05] qui permet de synthétiser des machines à états pour les objets participant dans des descriptions en langage UCM.

II.3. Langage UML, le standard de modélisation de l'OMG

Le langage UML (*Unified Modeling Language*) se définit comme le langage universel de modélisation graphique et textuelle. Il est né en début de 1997 dans l'optique d'unifier les méthodes d'analyse et de conception des systèmes. L'OMG adopte en novembre 1997 UML 1.1 comme langage de modélisation des systèmes d'information à objets. La version d'UML actuelle est UML 2.2 [OMG-UML] et les travaux d'amélioration se poursuivent.

Cette section sur UML est décomposée en trois sous-sections. La première donne une présentation de l'architecture du langage. La deuxième sous-section présente les mécanismes de communication utilisés par le langage UML. La trosième sous-section porte sur les diagrammes de spécification du comportement offerts par le langage, notamment les diagrammes d'interactions, d'activités et d'états.

II.3.1. Architecture d'UML

UML est un langage de modélisation qui fournit les fondements pour spécifier, construire, visualiser et décrire les artefacts d'un système logiciel. Afin d'assurer cela, UML se base sur une notation graphique dont la syntaxe est à la fois simple et intuitive. En fait, le choix de rendre UML un standard de modélisation permettant d'analyser/concevoir tout type de domaine a conduit à une sémantique vague et non précise. Cela fait sa force et parfois sa faiblesse.

Pour faciliter la définition et la formalisation d'UML, les différents concepts d'UML sont modélisés eux-mêmes en un sous-ensemble d'UML. Cette définition récursive, utilisée en méta-modélisation, décrit de manière semi-formelle les éléments de modélisation d'UML ainsi que la syntaxe et la sémantique de la notation qui permet de les manipuler. Les concepts d'UML sont groupés dans trois parties majeures :

Partie I – Structure. Cette partie définit les éléments de modélisation statiques et structurels (par exemple, classes et composants) utilisés dans divers diagrammes structuraux, tels que les diagrammes de classes, les diagrammes de composants, et les diagrammes de déploiement.

Partie II – Comportement. Cette partie définit les constructions dynamiques et comportementales (par exemple, la notion d'activité, d'interaction, et d'états/transitions) utilisées dans divers diagrammes comportementaux, tels que les diagrammes d'activités, les diagrammes de séquences, et les diagrammes de machine à états. Elle est organisée autour du package UML::CommonBehaviors. Ce dernier spécifie les concepts de base nécessaires pour décrire le comportement dans un système. Les deux sous-packages principaux de CommonBehaviors sont : BasicBehaviors et Communications. Le premier sous-package fournit les concepts nécessaires à la spécification du comportement. Les sous-types qui découlent de ce package sont des éléments concrets proposant différents mécanismes pour la définition du comportement. Le deuxième sous-package complète le premier en rajoutant les mécanismes nécessaires pour invoquer des comportements et établir des communications entre les objets du système.

Partie III – Supplémentaire (*Supplement*). Cette partie définit des éléments auxiliaires (par exemple, les types primitifs et les templates) et définit également la notion de profil employé pour adapter UML à différents domaines et plateformes.

II.3.2. Mécanismes de communication dans UML

UML dispose d'un ensemble de mécanismes de communication inter-objets. Chaque classificateur possède des *caractéristiques* (*Features*) qui composent sa signature. Ces caractéristiques se divisent en deux catégories : structurelles et comportementales. Les caractéristiques structurelles (*Structural Features*) sont décrites par les attributs du classificateur. Les caractéristiques comportementales (*Behavioral Features*) définissent les requêtes auxquelles les instances de ce classificateur seront capables de répondre. Parmi les mécanismes de communication proposés par UML on trouve [Le Guennec 01] :

- **Accès aux Attributs.** En rendant certains de ses attributs publics, une classe permet à tout objet du système d'accéder directement à ces attributs. Ce qui permet une communication implicite par partage de données.
- **Les opérations.** Chaque classe définit un certain nombre d'opérations que ses clients peuvent appeler. Comme toutes les requêtes que peut recevoir un objet, un appel d'opération est accompagné d'arguments dont les valeurs respectives sont liées aux paramètres formels de l'opération. Les appels d'opérations sont "synchrones", c'est-à-dire qu'ils ne passent pas par la file de messages éventuellement attachée à l'objet récepteur de l'appel si celui-ci est *actif*.
- **Les signaux.** Une classe *active* peut aussi définir des récepteurs (*Reception*) pour certains signaux, déclarant ainsi que les objets de cette classe peuvent recevoir les signaux référencés. La réception d'un signal par un objet se traduit par un événement-signal (*SignalEvent*) déposé dans la file d'attente de l'objet pour traitement ultérieur. Le client n'est pas bloqué lors de l'émission et peut poursuivre sans attendre son exécution. L'envoi de signal est donc un mécanisme de communication "asynchrone" et non-bloquant. C'est le mécanisme le plus souvent adopté pour la communication avec un objet actif.

II.3.3. Spécification du comportement dans UML

UML est destiné à décrire et représenter des besoins, spécifier et documenter des systèmes, et de concevoir des solutions. Il unifie à la fois les notations et les concepts orientés objet.

Dans sa version actuelle, UML 2.2 s'articule autour de treize types de diagrammes, chacun d'eux étant dédié à la représentation de concepts particuliers d'un système logiciel. Ces types de diagrammes sont répartis en deux groupes, les diagrammes structurels et les diagrammes comportementaux (cf. Figure II.6).

Diagrammes structurels. Les diagrammes structurels rassemblent les diagrammes suivants :

- diagramme de classes (*Class diagram*)
- diagramme d'objets (*Object diagram*)
- diagramme de composants (*Component diagram*)
- diagramme de déploiement (*Deployment diagram*)
- diagramme de paquetages (*Package diagram*)
- diagramme de structures composites (*Composite structure diagram*)

Diagrammes comportementaux. Les diagrammes comportementaux rassemblent les diagrammes suivants :

- diagramme de cas d'utilisation (*Use case diagram*)
- diagramme d'activités (*Activity diagram*)
- diagramme d'états-transitions (*State machine diagram*)
- diagramme de séquence (*Sequence diagram*)
- diagramme de communication (*Communication diagram*)
- diagramme global d'interaction (*Interaction overview diagram*)
- diagramme de temps (*Timing diagram*)

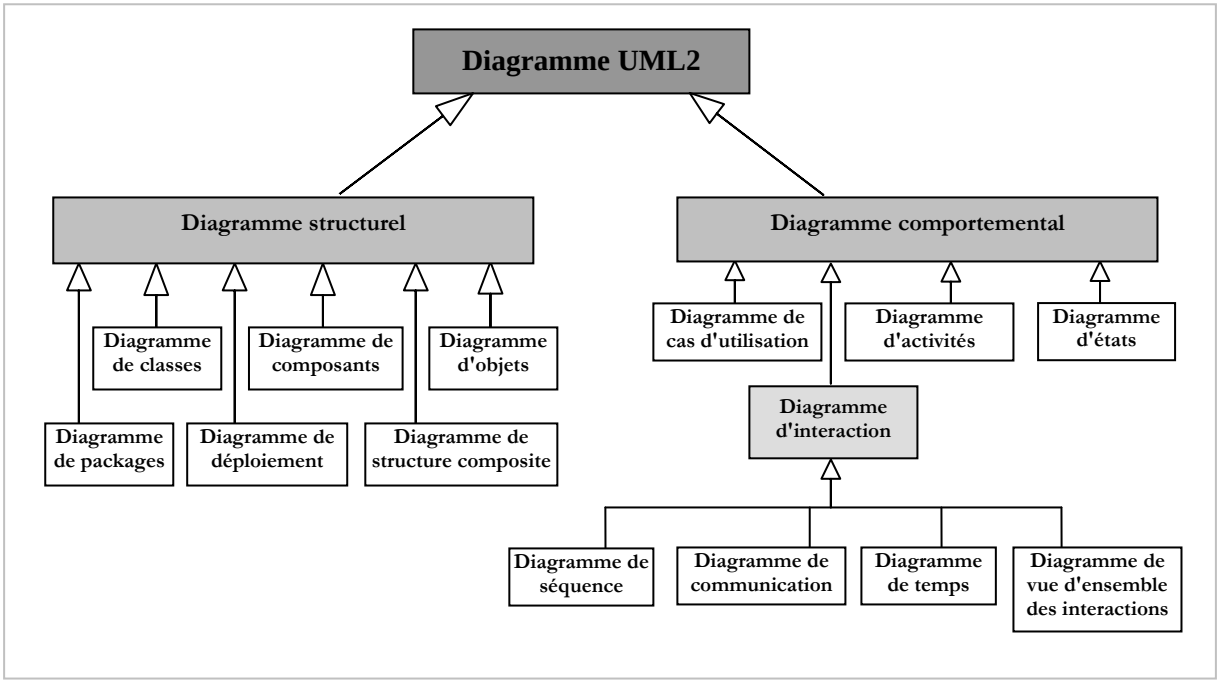


Figure II.6. Les diagrammes d'UML2 (version 2.2)

Ces diagrammes ne sont pas nécessairement tous produits lors d'une modélisation. Les plus utilisés sont généralement les diagrammes de cas d'utilisation, de séquence, de classes, d'objets, d'activités et d'états-transitions. Les diagrammes de composants et de déploiement sont surtout utiles pour la mise en œuvre où ils permettent de traiter les contraintes de la réalisation technique.

Dans cette section sur le langage UML, nous présentons dans un premier temps les mécanismes offerts par UML2 pour le traitement des interactions inter-objets, puis nous décrivons le mécanisme UML2 pour le traitement du comportement individuel des objets par des machines à états.

II.3.3.1. Diagrammes d'interaction

L'un des objectifs fondamentaux d'UML est de permettre aux utilisateurs de décrire non seulement des relations structurelles, mais aussi des processus et des séquences événementielles. Les diagrammes d'interaction ont pour but de décrire la communication entre objets. Ils établissent un

lien entre les diagrammes de cas d'utilisation et les diagrammes de classes en montrant comment des objets (des instances de classes) communiquent pour réaliser une certaine fonctionnalité. Les diagrammes proposés par UML pour décrire les interactions dans un système sont :

Diagramme de séquence : il permet une représentation séquentielle du déroulement des traitements et des interactions entre les éléments du système et/ou de ses acteurs. Le diagramme de séquence est une représentation intuitive lorsqu'on souhaite concrétiser des interactions entre deux entités (deux sous-systèmes ou deux classes d'un futur logiciel). Il permet à l'architecte de créer au fur et à mesure sa solution. Cette représentation intuitive est également un excellent vecteur de communication dans une équipe d'ingénierie pour discuter cette solution. Les diagrammes de séquence UML2 étendent largement les versions précédentes proposées dans UML1.x. Les diagrammes de séquence UML1.x décrivent un nombre fini d'interactions entre un ensemble d'objets sous forme de messages. Dans UML2, ces diagrammes sont considérés comme des collections d'événements, qui donnent la possibilité d'exprimer des interactions concurrentes et asynchrones. Cette extension est basée en grande partie sur le langage MSC (cf. section II.2.1.1).

Ainsi, les interactions dans UML2 peuvent se composer au moyen d'opérateurs pour obtenir des interactions beaucoup plus complexes. Les principaux opérateurs sont :

- Les opérateurs de choix et de boucle : alt (alternative), opt (option), break et loop ;
- Les opérateurs contrôlant l'envoi en parallèle de messages : parallel et critical region ;
- Les opérateurs contrôlant l'envoi de messages : ignore, consider, assertion et negative ;
- Les opérateurs fixant l'ordre d'envoi des messages : weak sequencing, strict sequencing.

La Figure II.7 illustre l'utilisation des deux opérateurs loop et alt. Cet exemple modélise l'envoi, par un enseignant, des notes d'une classe d'étudiants à l'administration et le traitement effectué sur ces notes.

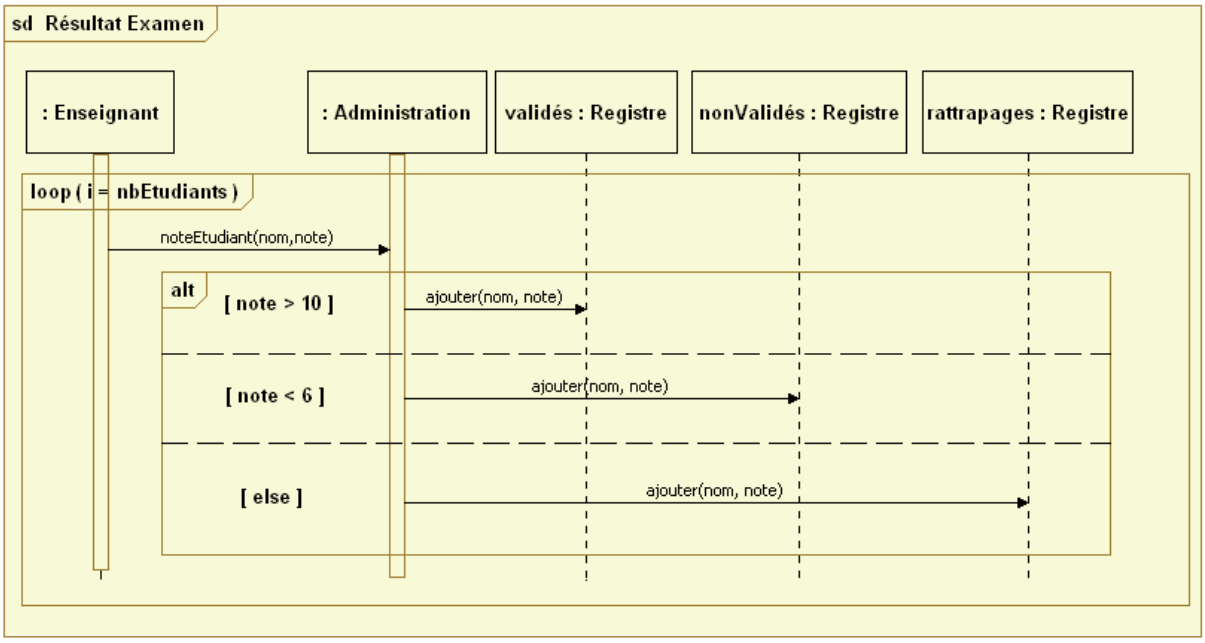


Figure II.7. Exemple d'utilisation des opérateurs loop et alt dans un diagramme de séquence

Diagramme de communication : appelé diagramme de collaboration en UML 1, c'est une représentation simplifiée d'un diagramme de séquence se concentrant sur les échanges de messages entre les objets. En fait, le diagramme de séquence et le diagramme de communication sont deux vues différentes mais logiquement équivalentes. C'est un graphe dont les nœuds sont des objets et les arcs (numérotés selon la chronologie) représentent les échanges entre objets.

Diagramme global d'interaction : permet de décrire les enchaînements possibles entre les scénarios préalablement identifiés sous forme de diagrammes de séquences. Les diagrammes globaux d'interaction définissent des interactions par une variante des diagrammes d'activité, d'une manière permettant une vue d'ensemble de flux de contrôle. Les lignes de vie et les messages n'apparaissent pas à ce niveau de vue d'ensemble.

II.3.3.2. Diagrammes d'activités

UML permet de représenter graphiquement le déroulement d'un cas d'utilisation ou le comportement d'une méthode, à l'aide de diagrammes d'activités. En effet, leur représentation sous forme d'organigrammes les rend facilement intelligibles et particulièrement adaptés à la description des cas d'utilisation. Ils montrent graphiquement le comportement d'une méthode et l'enchaînement des actions et décisions des acteurs du système dans le cadre d'un processus métier. Nous donnons ici les éléments de base constituant un tel diagramme.

Transition : Le passage d'une activité vers une autre est matérialisé par une transition. Les transitions sont déclenchées par la fin d'une activité et provoquent le début immédiat d'une autre.

Couloirs d'activités : UML permet, à l'aide de la notion de couloirs d'activités, d'organiser un diagramme d'activités selon les différents responsables des actions représentées. Il est possible d'identifier les objets principaux, qui sont manipulés d'activités en activités et de visualiser leur changement d'état.

Synchronisation : Il est possible de synchroniser les transitions à l'aide des barres de synchronisation *fork* et *join*. Une barre de synchronisation permet d'ouvrir et de fermer des branches parallèles au sein d'un flot d'exécution : les transitions qui partent d'une barre de synchronisation ont lieu en même temps ; et on ne franchit une barre de synchronisation qu'après réalisation de toutes les transitions qui sont rattachées à cette dernière.

La Figure II.8 illustre les principaux éléments utilisés dans les diagrammes d'activités. Cet exemple modélise l'enchainement des activités depuis le passage d'une commande jusqu'à la récupération de cette dernière par le client.

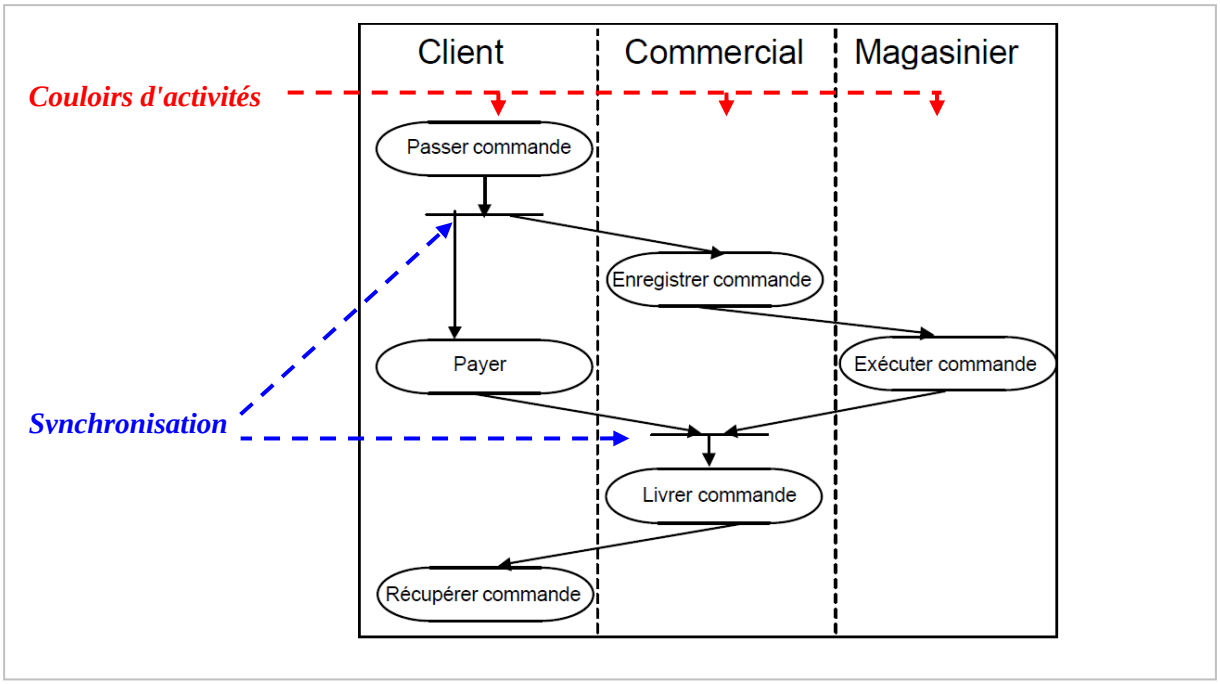


Figure II.8. Exemple d'utilisation du diagramme d'activités

II.3.3.3. Diagrammes d'états

UML a repris le concept bien connu de "machine à états finis" [Harel, 87], qui consiste à s'intéresser au cycle de vie des instances d'une classe particulière au fil de leurs interactions avec les autres entités du système, dans tous les cas possibles. Cette vue locale d'un objet décrit comment il réagit à des événements en fonction de son état courant et comment il passe dans un nouvel état.

Toutes les classes du modèle statique ne requièrent pas nécessairement une machine à états. Il s'agit donc de trouver celles qui ont un comportement dynamique complexe nécessitant une description poussée (les classes réactives). Cela correspond à l'un des cas suivants : (i) les objets de la classe peuvent réagir différemment à l'occurrence du même événement et chaque type de réaction caractérise un état particulier ; (ii) la classe doit organiser certaines opérations dans un ordre précis, et dans ce cas, des états séquentiels permettent de préciser la chronologie forcée des événements d'activation.

Une machine à états est souvent associée à un classificateur, par exemple une classe ou un sous-système. Cependant, UML ne définit pas de notation spécifique pour matérialiser cette relation. Une notation possible consiste à employer une note étiquetée avec le nom de la machine d'états et liée au classificateur. Une machine à états peut également être associée à une opération, décrivant ainsi son déroulement.

UML définit trois types d'états : (1) états simples ; (2) états composites, possédant une ou plusieurs régions correspondant à des sous-états ; (3) états sous-machine, sémantiquement équivalents aux états composites, mais destinés à regrouper des états dans un but de réutilisation. Un état peut posséder des activités qui sont déclenchées une fois ce dernier actif. On distingue trois types

d'activités qui sont : Entry, Exit et do. Ces activités sont déclenchées respectivement, lorsqu'on entre dans un état, lorsqu'on quitte un état, ou s'exécute continûment tant que l'état est actif.

Concernant les événements, UML propose de distinguer quatre sortes d'événements : (1) la réception d'un message envoyé par un objet, ou par un acteur ; (2) l'appel d'une opération sur l'objet récepteur ; (3) le passage du temps (time event) qui se modélise en utilisant le mot-clé *after* suivi d'une expression représentant une durée, décompté à partir de l'entrée dans l'état courant ; et (4) un changement dans la satisfaction d'une condition.

Une transition représente la relation entre deux états/pseudo-états. Elle modélise le changement effectif dans la configuration d'une machine à états lorsque celle-ci bascule d'un état à un autre. Une transition est spécifiée par la syntaxe suivante : `déclencheur[guarde]/effet`.

II.4. Conclusion

Ce premier chapitre de l'état de l'art nous a permis de présenter une étude des langages de modélisation utilisés pour la description du comportement. La première partie de ce chapitre a été dédiée à l'étude des langages représentatifs de ce domaine. Cette étude a été effectuée en se basant sur deux perspectives : langages de description à base de scénario, et langages de description à base d'état. Dans la deuxième partie, nous avons présenté le langage UML et avons décrit les formalismes intégrés à ce langage pour la description du comportement tels que les modèles d'interaction, les modèles d'activités et les machines à états.

D'après cette étude, on constate qu'il s'avère difficile de réunir tous les aspects de la modélisation comportementale au sein d'un même langage. D'une part, chacun des langages traités dans ce chapitre ne couvre qu'une perspective particulière du traitement du comportement. D'autre part, la complexité des systèmes logiciels évoluant constamment, les langages de modélisation ne peuvent pas anticiper cette évolution. Cela donne généralement lieu à l'apparition d'extensions de ces langages pour couvrir les nouveaux besoins. C'est le cas par exemple du langage MSC traité dans la section II.2.1.1. Cette évolution de la complexité peut donner naissance également à des variantes multiples d'un même langage. On peut citer le cas du langage Statecharts traité dans la section II.2.2.1 où le travail de Van der Beek [Beek, 94] donne un aperçu de vingt variétés de ce langage, chacune ayant sa propre sémantique.

Le langage UML se définit comme le langage universel de modélisation graphique et textuelle. Il a bénéficié de la réussite de langages spécialisés dans le traitement de certaines perspectives de la modélisation pour couvrir l'analyse/conception des systèmes à objets. Par exemple, les diagrammes de séquence UML ont été inspirés des constructions du langage MSC. Ou bien encore, les diagrammes d'états UML ont adopté des principes fondamentaux de Rhapsody (variante des statecharts). Dans sa version actuelle, UML 2.2 s'articule autour de treize types de diagrammes, chacun d'eux étant dédié à la représentation de perspectives particulièrs d'un système logiciel. Cette diversité de diagrammes répond aux suggestions de la communauté de converger vers un langage de modélisation standard qui couvre l'ensemble des domaines d'application.

Afin d'assurer cela, UML se base sur une notation graphique dont la syntaxe est à la fois simple et intuitive. Le choix de faire d'UML un standard permettant d'analyser/concevoir tout type de système a conduit à une sémantique vague et non précise. Cela en fait sa force et parfois sa faiblesse.

La version d'UML actuelle est UML 2.2 [OMG-UML] et les travaux d'amélioration et d'extension se poursuivent.

Le contexte applicatif de notre travail de thèse se base sur le langage UML. L'objectif global est de proposer une extension du profil VUML (profil UML), pour la modélisation des aspects comportementaux dans une conception multi-modèles orientée points de vue. Certes, nous allons tirer profit de la réussite du langage UML et les mécanismes engagés pour le traitement du comportement, par contre la sémantique faible donnée à ces mécanismes constitue un problème à considérer, surtout dans un contexte de multi-modélisation.

L'OMG préconise l'utilisation du langage UML comme langage de description de système basé sur les modèles. Plusieurs approches de modélisation multi-modèles à base d'aspects, de points de vue, de sujet, etc., ont adopté le standard UML comme moyen pour décrire les modèles de conception. La problématique centrale dans ce genre d'approches (incluant l'approche par points de vue) se focalise sur la composition des modèles partiels issus d'une modélisation séparée des préoccupations.

Dans le chapitre suivant, nous passons en revue les principales approches de multi-modélisation, et décrivons les mécanismes de composition associés.

CHAPITRE III. MULTI-MODELISATION ET VUML

Sommaire

III.1. Introduction

La modélisation des systèmes logiciels est un domaine de recherche dans lequel les approches basées sur la séparation des préoccupations ont beaucoup d'intérêt. En effet, les exigences actuelles en modélisation des systèmes se complexifiant, la construction d'un modèle global prenant en compte simultanément tous les besoins est extrêmement difficile, voire impossible dans certains cas. Plusieurs facteurs sont la cause de cette évolution de la complexité :

- Les besoins fonctionnels sont en augmentation permanente : les tendances convergent vers des systèmes garantissant la fiabilité, la performance et une durée de vie maximale. Les systèmes doivent être le plus complets possible en répondant aux exigences fonctionnelles actuelles et anticiper sur les exigences futures, etc.
- Les besoins transversaux et techniques sont en augmentation afin d'intégrer la sécurité et la robustesse, de garantir l'intégrité des données, de permettre la portabilité, etc.
- Les utilisateurs d'un système (par exemple : clients, personnels, administrateurs) sont souvent multiples, avec des besoins spécifiques (profils) et des attentes particulières envers le système. Les systèmes doivent être suffisamment flexibles pour leur offrir des accès personnalisés.

Avec cet accroissement de la complexité des systèmes logiciels, on se trouve amené à gérer des composants hétérogènes, chacun répondant à plusieurs préoccupations à la fois. Une conséquence évidente de cette complexité est la perte de modularité dans les systèmes développés, ce qui freine leur évolution et leur réutilisation. Ce problème est caractérisé par des symptômes connus sous les noms de dispersion (*scattering*) et couplage de fonctionnalités (*tangling*), (identifiés pour la première fois par [Kiczales, 97] et servant de justification pour l'approche de programmation par aspects).

Scattering : c'est le cas où une préoccupation (fonctionnelle ou transversale) est distribuée dans tout le système, et non pas placée dans une unité bien identifiée. Par exemple, si une fonctionnalité est distribuée sur plusieurs composants, le coût d'une mise à jour de cette fonctionnalité peut être considérable.

Tangling : c'est le cas où une unité contient des éléments provenant de différentes préoccupations. Dans ce cas, nous avons dans un même composant une inter-liaison entre plusieurs fonctionnalités, et par conséquent le coût de gestion des différentes interactions entre ces fonctionnalités peut être important.

Pour pallier cette situation, plusieurs approches adoptant le principe de séparation des préoccupations ont été proposées. Elles permettent une décomposition de la modélisation, notamment dans la phase de conception où plusieurs modèles peuvent être développés séparément pour représenter une perspective particulière du système. Les modèles partiels développés doivent être ensuite composés pour produire le modèle final du système.

Ce chapitre est organisé en trois parties. Dans la première partie, nous passons d'abord en revue certaines techniques de multi-modélisation comme la modélisation par aspects et la modélisation par sujets, qui répondent à des problèmes comparables à ceux que nous traitons dans le cadre orienté points de vue. Nous décrivons dans la deuxième partie un aperçu des approches de composition de modèles, notamment celles à base d'aspects qui sont les plus proches de notre problématique. Cette partie est conclue par une synthèse récapitulative. Nous consacrons la troisième

partie à la description du contexte applicatif de la thèse qui est le langage VUML (*View based Unified Modelling Language*) ainsi que la démarche de conception associée [Anwar, 09]. Nous rappelons les concepts de VUML qui concernent principalement la modélisation par points de vue de la structure d'un système. A la fin du chapitre nous discutons les limites actuelles de ce langage et nous introduisons la problématique de la spécification comportementale par points de vue.

III.2. Spécification multi-modèle de système

Lors du développement d'un système complexe, la construction d'un modèle global prenant en compte simultanément tous les besoins des acteurs et les exigences techniques imposées est souvent impossible. Malgré l'évolution des techniques d'analyse/conception dans le domaine du génie logiciel, la construction de systèmes informatiques reste une tâche difficile. Dans la réalité, soit plusieurs modèles partiels sont développés séparément et coexistent avec les risques d'incohérence associés, soit le modèle global doit être fréquemment remis en cause quand les besoins évoluent.

L'approche objet et les concepts associés (encapsulation, héritage, polymorphisme) ont constitué une avancée importante pour la conception de systèmes logiciels en apportant notamment la modularité et la réutilisation. Mais cette approche présente des limites lorsqu'il s'agit de faire face à des systèmes complexes, multidimensionnels (i.e. systèmes à plusieurs niveaux : fonctionnel, métier, technologique, etc.) et hautement parallèles.

Pour maîtriser cette complexité, on utilise de plus en plus des approches de modélisation dites *multi-modèles*. En effet, la multi-modélisation permet une décomposition de la modélisation par facettes, notamment dans la phase de conception où plusieurs modèles peuvent être développés séparément pour représenter une perspective particulière du système. Les modèles partiels développés sont ensuite composés (tissés) pour produire le modèle final du système. La multi-modélisation peut être conçue de différentes manières : programmation par sujets [Harrison et al., 93], programmation par rôles [Kristensen et al., 96], programmation par aspects [Kiczales et al., 97] et programmation par points de vue [Kriouile, 95 ; Coulette et al., 96 ; Muller et al., 03]. Dans cette section, nous passons en revue ces approches.

III.2.1. Modélisation par points de vue

Un point de vue est défini dans le dictionnaire ROBERT par : "un endroit où l'on doit se placer pour voir un objet le mieux possible ou encore comme étant une manière particulière dont une question peut être considérée". Les termes les plus proches de la définition du point de vue sont : aspect, optique, perspective et vue. En informatique, cette notion de point de vue possède une multitude de significations qui divergent selon les travaux et les domaines. Les concepts de vue et de point de vue ont été étudiés dans plusieurs domaines liés au traitement de l'information : bases de données, représentation des connaissances, analyse et conception, langages de programmation, outils de génie logiciel, etc.

Dans le domaine des bases de données, la notion de vue est exploitée par les langages d'interrogation comme une fonction de sélection sur les données [Abiteboul et al., 91 ; Debrauwer, 98]. En représentation des connaissances, les vues sont utilisées pour représenter les raisonnements classificatoires et la représentation taxinomique de la connaissance. Un point de vue

détermine un ensemble de caractéristiques liées à un concept ou à une famille d'objets. Un concept peut être observé selon différents points de vue (cf. TROPES [Marino 93] et SHOOD [Rieu et al., 92]).

Notre équipe de recherche travaille depuis plusieurs années sur l'intégration de la notion de point de vue dans l'analyse/conception de systèmes logiciels [Marcaillou, 95 ; Kriouile, 95 ; Coulette et al., 96]. Le travail de Nassar [Nassar, 05] a donné lieu à l'établissement d'un profil UML appelé VUML (*View based UML*), qui permet d'analyser/concevoir un système logiciel par une approche combinant objets et points de vue (cf. III.4).

III.2.2. Modélisation par aspect

La modélisation par aspects (*Aspect-Oriented Modeling ou AOM*) est une approche de multi-modélisation fondée sur la séparation entre les préoccupations fonctionnelles et les préoccupations dites « transversales » lors du développement logiciel. L'idée de la modélisation par aspects découle de la programmation par aspects et propose de considérer les aspects dans les modèles [Klein, 06]. Face à l'émergence de l'IDM, le paradigme aspect a été étendu aux phases amont du développement logiciel, c'est-à-dire à la phase de conception [Clarke, 01] ou encore à la phase d'analyse des exigences [Jacobson et al., 04].

Le terme d'*aspect* est associé aux fonctionnalités qu'on ne peut pas rassembler efficacement dans un seul module (par exemple : sécurité, performance, persistance, etc.). Le paradigme de la modélisation par aspects considère que les services de base d'une application (les préoccupations fonctionnelles) et ses propriétés ou fonctionnalités transversales, doivent être indépendants et découplés les uns des autres aussi bien à la conception qu'à l'implémentation. Par exemple, l'abstraction *Carte de crédit* admet une préoccupation fonctionnelle représentant le processus de paiement. Elle admet aussi d'autres préoccupations techniques (transversales) telles que la gestion de l'authentification lors de la connexion, l'intégrité de la transaction de paiement, etc.

La démarche suivie pour une modélisation à base d'aspects consiste à la décomposition de l'application en aspects et en unités de base. Une unité de base représente une fonctionnalité – ou une partie d'une fonctionnalité – d'un système. Un aspect représente une propriété transversale qui affecte les fonctionnalités de base d'une manière systématique. Ces propriétés peuvent être conçues et analysées séparément des fonctions de base de l'application. Il est possible ainsi de définir sur un modèle de base, mêlant différentes préoccupations transversales, autant d'aspects que nécessaire pour représenter ces diverses préoccupations.

III.2.3. Modélisation par sujet

La programmation par sujets ou SOP (*Subject Oriented Programming*) est une autre technique de séparation de préoccupations introduite par [Harrison et al., 93]. Cette approche est fondée sur une séparation multidimensionnelle des préoccupations, permettant de couvrir différents types de préoccupations (métier, technologiques, règles de gestion, etc.). Elle permet d'identifier un ensemble de spécifications et de comportements reflétant la perception du monde réel correspondant à une vision générique d'un acteur. Par exemple, un administrateur d'une formation a ses propres caractéristiques pour un *cours* : un prix et des méthodes de gestion. Ces caractéristiques peuvent

s'appliquer à n'importe quel objet à vendre : produit, immobilier, etc. Ces caractéristiques sont dites alors extrinsèques à l'objet "Cours". Elles font partie de la vue subjective générique de l'administrateur.

Un sujet ne correspond pas à une classe mais c'est une spécification de hiérarchie de classes. Cette hiérarchie dénote l'ensemble de descriptions d'objets pour une vision particulière. Cette spécification ne décrit pas de structure pour des instances objet mais c'est une description schématique qui peut s'appliquer à un domaine particulier d'objets [Tarr et al., 99]. Ce sont les instances d'un sujet, appelés activations de sujet, qui contiennent effectivement les données d'un système. Un sujet peut être activé pour plusieurs domaines et de son coté, un objet peut activer plusieurs sujets. Le lien entre les différentes activations est réalisé à travers la notion d'identité d'objet.

L'intégration de l'ensemble des sujets est déterminée par un ensemble de règles dites de composition [Ossher et al., 95]. Une règle de composition met en jeu deux ou plusieurs sujets destinés à être intégrés. On distingue deux catégories principales de règles de composition : les règles de correspondance et les règles de combinaison. Les règles de correspondance traitent les liens syntaxiques et sémantiques entre classes (respectivement propriétés des classes) ayant des noms identiques ou différents, et appartenant à différents sujets destinés à être intégrés. Les règles de combinaison précisent la façon dont les classes (respectivement propriétés des classes) de sujets doivent être composées. La définition des règles de composition est encapsulée dans un ou plusieurs modules de composition, qui mettent en jeu plusieurs sujets à intégrer [Ossher et al., 01].

III.2.4. Modélisation par rôle

Dans un système complexe, un objet interagit avec le monde de manière subjective selon la perspective du contexte de l'utilisateur [Harrison et al., 93]. L'apparition de la notion de rôle vient du fait que les propriétés extrinsèques d'un objet peuvent changer dans le temps [Pernici, 90]. En effet, un objet peut être sujet à des classifications multiples durant son cycle de vie, et à chaque fois il joue un rôle adapté à une situation particulière. Kristensen définit un rôle comme étant un point de vue temporaire [Kristensen et al., 96]. Riehle affine cette définition de rôle en lui associant un type qui décrit la vue qu'a un objet vis à vis d'un autre et en précisant qu'un objet peut jouer différents rôles à un instant donné [Riehle et al., 98].

En résumé, le concept de rôle aborde trois questions principales qui surviennent lorsqu'on veut modéliser l'aspect dynamique des entités avec les modèles d'objet traditionnels [Dahchour et al., 06] :

- **Le changement dynamique de classe** : les objets changent de classification. Par exemple, dans le temps, une *personne* cesse d'être un étudiant, et devient un lauréat. Deux cas différents surviennent selon que l'objet qui subit la transition est conservé comme une instance de la classe source ou pas :
  - **une extension** décrit le cas où l'objet reste une instance de la classe source.
  - **une évolution** décrit le cas où l'objet n'est plus une instance de la classe source.
- **L'instanciation multiple d'une même classe :** un objet peut être instance plus d'une fois de la même classe. Par exemple, un étudiant peut être inscrit dans deux universités différentes. Il jouera donc deux rôles différents.

- **L'accès contextuel :** la possibilité de voir un objet multi-facettes dans une perspective particulière. Par exemple, la personne peut être vue séparément comme un employé ou comme un étudiant.

Dans la littérature, il existe peu de travaux qui traitent cette approche de modélisation par rôles. Dans une étude antérieure à cette thèse, nous avons travaillé sur la possibilité d'extension du langage UML par cette notion de rôle. Le résultat de ce travail est détaillé dans [Dahchour et al., 06] et [Dahchour et al., 07].

III.2.5. Synthèse

Comme nous l'avons décrit dans les sous-sections précédentes, plusieurs approches ont abordé la notion de *vue* sous différents termes : vue bien sûr, mais aussi sujet, rôle et aspect. Elles sont toutes fondées sur un ensemble de principes communs, chacune trouvant sa particularité dans les critères de décomposition des différentes préoccupations.

L'approche par aspects [Kiczales et al., 97] décompose le système en unités fonctionnelles et en unités non fonctionnelles. Elle sépare ainsi les fonctionnalités de base (ou métiers) d'une application des fonctionnalités transversales extrinsèques aux exigences métiers. Dans l'approche par points de vue [Finkelsetin et al., 92 ; Kriouile, 95 ; Mili et al., 01], le niveau de décomposition est différent de celui adopté par l'approche par aspects. Il s'agit d'une décomposition suivant les points de vue des acteurs du système. Les vues sont développées indépendamment les unes des autres et sans faire de distinction entre les fonctionnalités de base et les fonctionnalités transversales. Le résultat de ce type de décomposition est un ensemble d'entités décrites selon les vues subjectives des acteurs du système. L'approche par sujet [Ossher et al., 95] étendue par l'approche MDSoC (*Multidimensional Separation of Concerns*) [Tarr et al., 99], propose une décomposition du système en plusieurs dimensions arbitraires, ou chaque dimension regroupe un ensemble de préoccupations particulières. On parle de préoccupation au sens large, sans différencier entre préoccupation de base ou préoccupation transversale. Dans l'approche par rôle [Pernici, 90 ; Kristensen et al., 96], il s'agit de représenter une entité du modèle par l'intermédiaire de plusieurs objets. Chaque objet modélise un rôle particulier joué par cette entité. A la différence de la modélisation par points de vue, les rôles sont des objets résultant de vues subjectives locales aux entités sans qu'ils soient liés aux acteurs du système. Les différentes vues subjectives du système représentent l'ensemble des préoccupations de l'application, et chaque préoccupation est représentée par les différents rôles et leurs interactions. Chaque groupe d'objets représente une préoccupation au sens large sans faire de distinction entre préoccupation de base et préoccupation transversale.

L'utilisation d'une approche basée sur le principe de la séparation des préoccupations présente un certain nombre d'avantages pour le cycle du développement logiciel, de l'analyse des besoins, en passant par la conception et le développement, jusqu'à l'exploitation et la maintenance. Parmi ces avantages on peut citer :

- Offrir la possibilité d'un développement décentralisé, où plusieurs équipes de développeurs peuvent travailler librement au sein d'un même projet ;
- Apporter une bonne modularité à l'application, chaque préoccupation (fonctionnelle ou non) du système étant définie, conçue et encapsulée proprement. Cela se répercute naturellement

sur l'ensemble du cycle de développement, et par conséquent on obtient à la fin un code propre, lisible, facile à comprendre, à faire évoluer et à réutiliser ;

- Permettre de faire évoluer plus facilement les fonctionnalités d'un système. De telles approches donnent plus de souplesse en cas de mise à jour d'une fonctionnalité, et rendent possible l'ajout de nouvelles préoccupations non prévues a priori.

Cependant, plus la phase de décomposition ainsi que la phase de développement des préoccupations appliquent le principe de séparation et d'indépendance des préoccupations, plus la phase d'assemblage du système final devient délicate. En effet, la décomposition du système en modèles partiels n'est qu'une première phase dans la démarche de développement de ces approches. La deuxième phase, incontournable, de la démarche est la phase de composition des modèles partiels produits. De nombreux travaux de recherche s'intéressent à cette problématique et chacun de ces travaux propose sa propre vision du sujet. La section suivante est dédiée au traitement de la problématique de composition de modèles.

III.3. Composition de modèles

Dans un processus de développement à base de modèles, la composition consiste à combiner un certain nombre de modèles pour en créer un ou plusieurs. La composition apparaît sous divers termes selon le contexte d'application. Dans le domaine des bases de données [Batini et al., 86 ; Navathe et al., 86], la composition se manifeste par l'opération d'**intégration** d'un ensemble de vues d'une base de données, ou de plusieurs schémas de bases de données hétérogènes et réparties. En ingénierie des exigences, les préoccupations du système sont souvent prises en compte selon des points de vue différents [Finkelstein et al., 90]. Il en résulte un ensemble de modèles appelés aussi des vues. L'obtention d'une vue globale sur le système est réalisée à travers la **fusion** des vues partielles. Dans le développement de logiciels par aspects (Aspect Oriented Software Development- AOSD) [Kiczales et al., 97 ; France et al., 04a ; Baniassad et al., 04] on parle souvent d'opération de **tissage**.

Nous avons structuré cette section en deux parties. Dans la première, nous présentons un nouveau concept de fusion, proposé par UML dans sa version 2.0, appelé *PackageMerge* [OMG, 03a]. Dans la deuxième partie, nous proposons un survol des techniques de composition de modèles dans les approches de multi-modélisation.

III.3.1. Composition de modèles dans UML

UML2.0 propose un mécanisme de fusion de paquetages appelé *PackageMerge* [UML-OMG]. *PackageMerge* est une relation dirigée entre deux paquetages indiquant que le contenu de ces deux paquetages doit être combiné. Le principe est similaire au concept de relation de généralisation au sens où l'élément source ajoute les caractéristiques de l'élément cible à ses propres caractéristiques. Ce mécanisme est utilisé lorsque des éléments définis dans différents paquetages ont le même nom et représentent le même concept. Il est utilisé le plus souvent pour fournir des définitions différentes d'un même concept destiné à être utilisé différemment, à partir d'une définition commune de base.

La relation *PackageMerge* diffère de l'importation de paquetage au sens où il y a création de relations entre des classes de même nom. Par exemple, UML peut définir le concept de relation *include* à un niveau générique, et le spécialiser pour différents contextes d'utilisation tout en conservant son

nom. Conceptuellement, l'effet de la relation *PackageMerge* peut être considéré comme une opération qui prend le contenu de deux paquetages et produit un nouveau paquetage combinant le contenu des deux paquetages. La Figure III.1-a illustre ce principe.

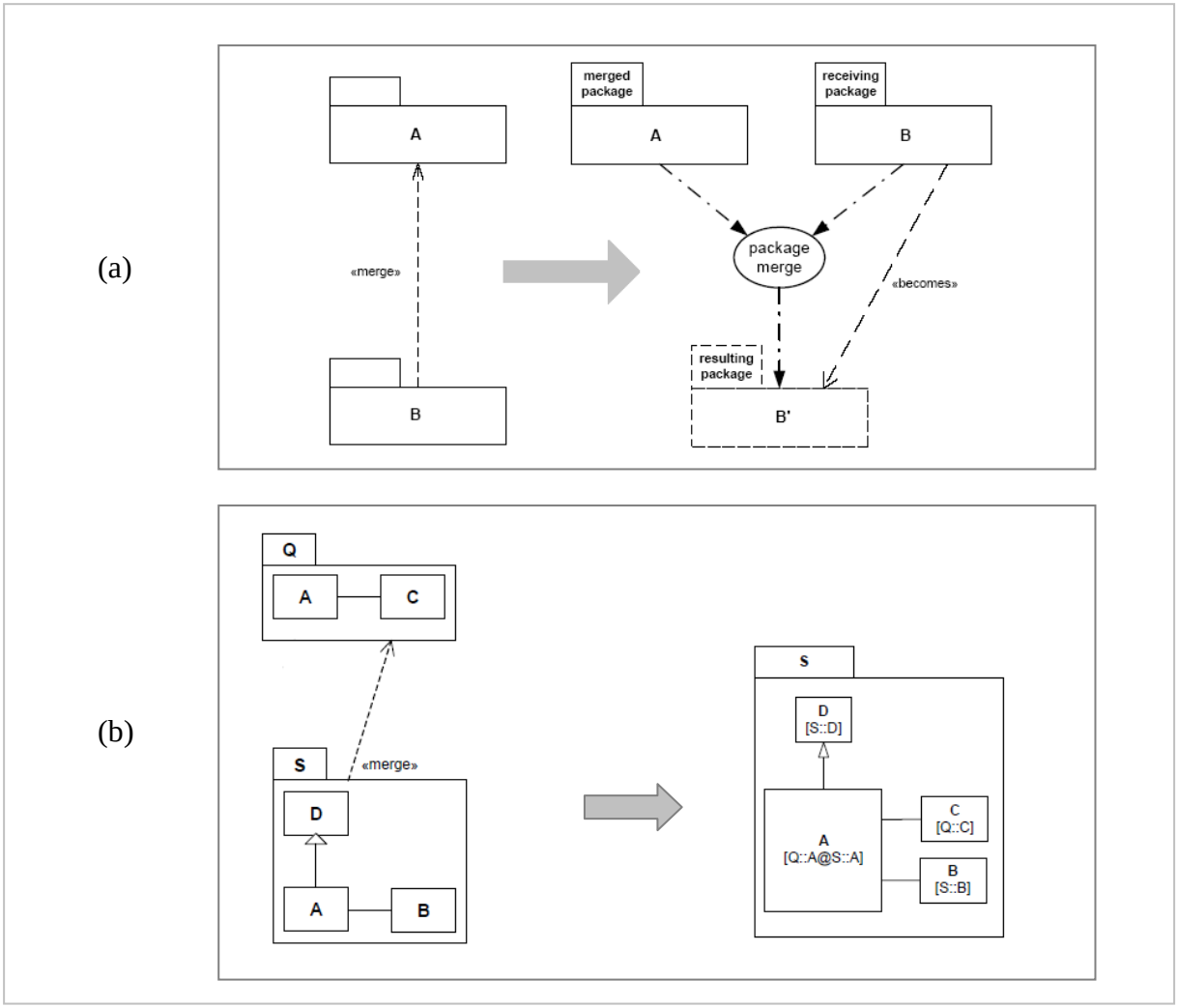


Figure III.1. Illustration du principe de la relation PackageMerge

La relation *PackageMerge* implique un ensemble de transformations. Ces dernières spécifient la sémantique de la fusion entre deux paquetages reliés par une telle relation comme montré par la Figure III.1-a. Le paquetage source de la relation *PackageMerge* (paquetage B) est appelé paquetage récepteur. C'est celui dont le contenu est étendu après l'opération de fusion. Le paquetage destination de la relation (paquetage A) est appelé paquetage fusionné. C'est le paquetage qui contient les éléments supplémentaires ajoutés au paquetage récepteur. La Figure III.1-b donne une illustration de ce principe.

Dans la spécification UML2 [OMG, 03a], la sémantique de la relation *PackageMerge* est définie par un ensemble de règles de transformations et de contraintes. Les règles de transformation assurent la mise en correspondance des éléments et la fusion des éléments correspondants. Ces règles sont définies pour chaque type de méta-classe. Les contraintes permettent d'assurer la validité de la fusion. Comme pour les règles de transformation, les contraintes sont spécifiées pour chaque type de méta-classe.

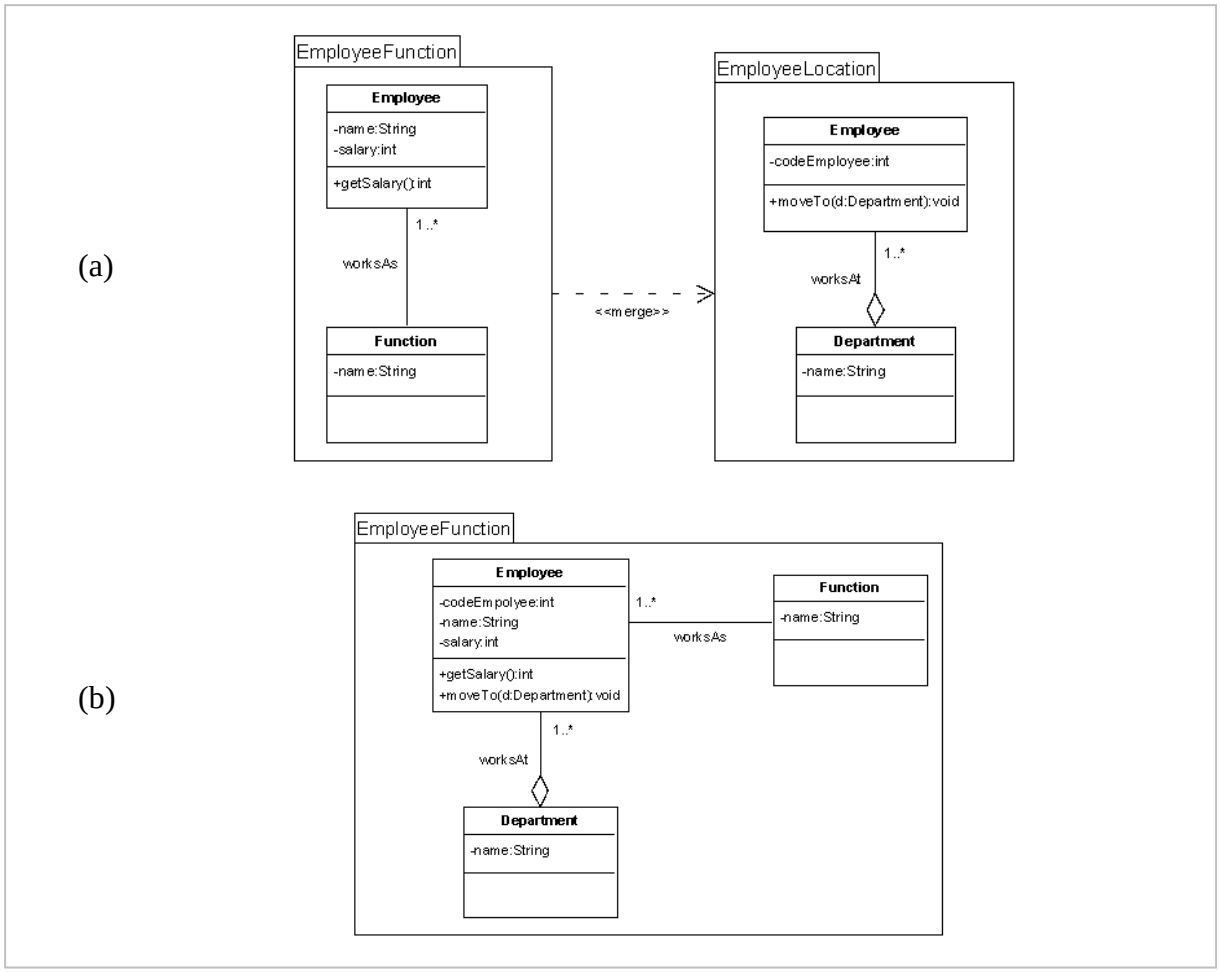


Figure III.2. Un exemple de fusion avec PackageMerge

La Figure III.2-a montre un exemple d'utilisation de la relation *PackageMerge* [Zito et al., 06]. Dans cet exemple, on souhaite étendre la définition du concept *Employee* du paquetage *EmployeeFunction* en rajoutant une information indiquant le département auquel il est rattaché. Le paquetage *EmployeeFunction*, source de la relation *PackageMerge*, joue le rôle de paquetage récepteur alors que le paquetage *EmployeeLocation* dans lequel le concept de département est défini, représente le paquetage fusionné. Le résultat de la fusion de ces deux paquetages donne lieu au paquetage résultant de la Figure III.2-b. Les éléments du paquetage *EmployeeFunction* résultant sont obtenus par fusion des éléments du paquetage fusionné avec les éléments du paquetage récepteur. En effet, la classe *Employee* du paquetage *EmployeeFunction* correspond à la classe du même nom dans le paquetage *EmployeeLocation* ; ces deux classes sont fusionnées en ajoutant l'attribut *codeEmployee* et l'opération *moveTo()* à la classe *Employee* du paquetage récepteur. La classe *Department* et l'association *worksAs* n'ayant pas de correspondant, elles sont simplement copiées dans le paquetage résultant.

La relation *PackageMerge* de UML2 propose une solution intéressante pour la fusion des paquetages à condition que les concepts définis dans ces paquetages aient une définition de base commune. C'est pourquoi elle est souvent utilisée en méta-modélisation, par exemple pour assurer la conformité entre les niveaux du méta-modèle UML2. Son intérêt est beaucoup moins évident quand les paquetages à fusionner ont une distance sémantique importante. En résumé, bien que cette

relation simplifie certains mécanismes internes liés à la méta-modélisation avec UML, son utilisation en pratique est assez limitée dans le monde du développement et des outils de modélisation.

III.3.2. Composition de modèles dans les approches de multi-modélisation

Comme nous l'avons vu dans la section III.2, la décomposition effectuée dans la phase amont de la démarche de multi-modélisation offre de nombreux avantages. Cependant, plus la phase de décomposition des préoccupations applique le principe de séparation et d'indépendance du développement, plus la phase d'assemblage du système final devient délicate. Aussi, chacune des approches de multi-modélisation considérées ci-dessus propose ses propres mécanismes de composition :

Approche par aspect : Le processus de composition, appelé tissage (*Weaving*), a lieu lors de la compilation, en respectant les points de jonction [Kiczales et al., 01]. Le tisseur d'aspects (*Aspect Weaver*) est un mécanisme de composition qui assure en fait la recomposition de l'application.

Approche par points de vue : Le processus de composition commence d'abord par la résolution des différents conflits en déterminant les incohérences et les similitudes entre les éléments constituant les modèles partiels. Une fois les conflits résolus, la composition s'effectue en appliquant un ensemble de règles de transformation que l'on peut décliner en règles de correspondance, de fusion et de translation [Anwar et al., 08a].

Approche par sujet : L'intégration des sujets est déterminée par un ensemble de règles de composition [Ossher et al., 95]. Toute règle de composition met en jeu deux ou plusieurs sujets (les classes qui les composent et leurs propriétés structurelles et comportementales) destinés à être intégrés. Les règles de composition sont de trois types : règles de mise en correspondance, de combinaison, ou de combinaison et de mise en correspondance en même temps. Les règles de mise en correspondance indiquent les correspondances, si elles existent, entre classes (respectivement propriétés des classes) ayant des noms identiques ou différents, et appartenant à différents sujets destinés à être intégrés. Les règles de combinaison précisent la façon dont les classes (respectivement propriétés des classes) de sujets doivent être composées (fusion, redéfinition, ou encore fusion avec un ordre d'exécution particulier). La définition des règles de composition est encapsulée dans un ou plusieurs modules de composition, qui mettent en jeu plusieurs sujets à intégrer ensemble.

Dans les sous-sections qui suivent, nous proposons un survol des techniques de composition de modèles issus du domaine de développement par aspects AOSD (*Aspect Oriented Software Development*). Nous avons privilégié les travaux de composition à base d'aspects car ce sont ceux qui sont les plus proches de notre problématique. Nous décrivons ainsi l'approche *Theme* de Clarke [Clarke, 02], [Baniassad et al., 04], *AAM* de France et al. [France et al., 04a ; Reddy et al., 06], *MATA* de Whittle [Whittle et al., 07a ; Whittle et al., 07b].

III.3.2.1. Composition de modèles dans l'approche Theme

Theme [Baniassad et al., 04 ; Clarke, 04] est une approche d'analyse/conception orientée aspect, définie comme une extension du langage UML pour supporter la modularisation des préoccupations d'un système, notamment les préoccupations transverses (transactions, distribution,

etc.). Le modèle de développement proposé par cette approche se situe à deux niveaux : le niveau d'identification des exigences (Theme\Doc) et le niveau de conception (Theme\UML). Theme\Doc propose un modèle à base d'un graphe d'analyse et un support pour la visualisation et la spécification des exigences d'un système. Ce niveau sert à documenter et à identifier les relations entre les fonctionnalités à concevoir, et à révéler les fonctionnalités du système considérées comme transverses. Theme\UML permet de modéliser les fonctionnalités et les aspects du système, et de spécifier comment ces thèmes doivent être composés. Dans la suite, nous détaillons en particulier l'approche Theme\UML et nous explicitons comment les modèles de conception (*Themes*) sont composés dans cette approche.

III.3.2.1.1. Aperçu de l'approche THEME\UML

Theme\UML a pour but de résoudre le problème de la dispersion des exigences dans les modèles de conception. Un *Theme* correspond à un élément de conception permettant l'encapsulation d'une fonctionnalité ou d'une préoccupation transverse. Il est représenté par un paquetage stéréotypé par « theme ». Il peut être combiné avec d'autres Themes à l'aide d'une relation de substitution appelée 'bind' qui exprime la composition entre deux Themes. Un Theme est représenté par un paquetage paramétré. Chaque paramètre correspond à une classe et à l'ensemble des méthodes sur lesquelles la fonctionnalité doit être tissée. La partie structurelle du système est souvent représentée dans Theme\UML par des diagrammes de classes, tandis que les diagrammes de séquence sont utilisés pour décrire le comportement.

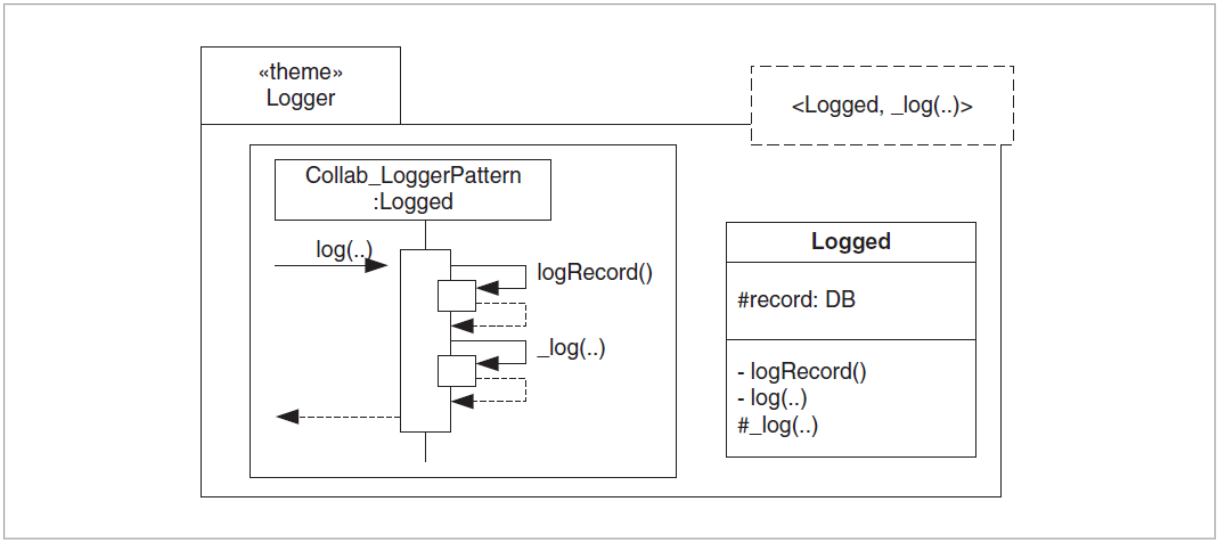


Figure III.3. Représentation de la fonctionnalité de Log dans Theme\UML [Baniassad et al., 04]

La Figure III.3 représente le Theme *Logger*, exemple classique de préoccupation transverse. Étant donné que toutes les opérations du système doivent être tracées, toutes les opérations des Themes doivent être complétées par un comportement d'authentification (*logging*). Cet aspect a été défini comme une fonctionnalité générique pour être en mesure de concevoir le comportement de logging séparément des opérations qui nécessitent une étape d'authentification. L'aspect *Logger* comporte deux paramètres de template : la classe *Logged* et la méthode *_log*. Le tissage du Theme de base *CMS* (Course Management System) avec le Theme *Logger* utilise une relation de composition de

type 'bind' (Figure III.4). Cette relation permet de substituer le paramètre *Logged* par les classes *Person*, *Student* et *Professor* définies dans CMS, et de substituer la méthode *_log* par leurs méthodes respectives *register*, *unregister* et *giveMark*.

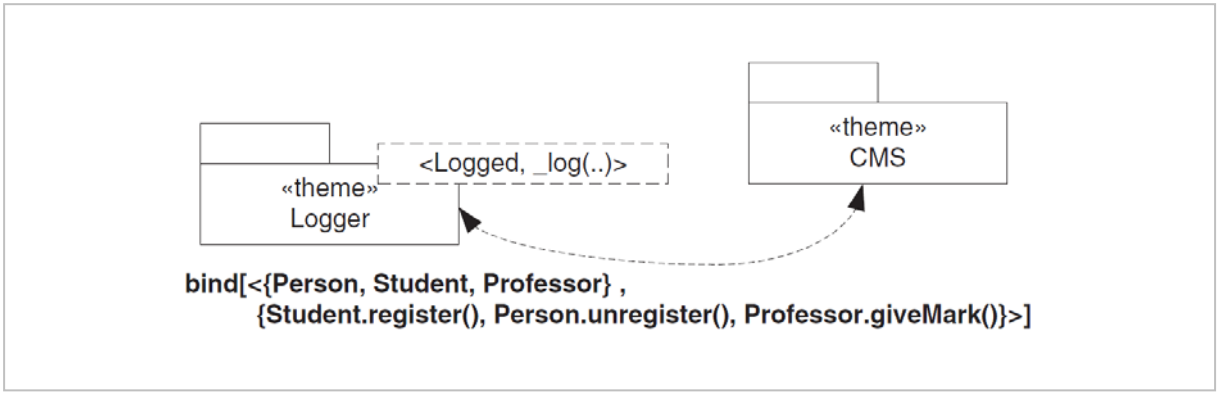


Figure III.4. Composition des Themes Logger et CMS [Baniassad et al., 04]

III.3.2.1.2. Technique de composition utilisée

La composition de modèles dans Theme\UML est spécifiée par une *relation de composition* qui permet d'identifier les parties identiques dans les *Themes* à composer, et de spécifier comment ces parties doivent être composées. Trois types de composition sont possibles : (i) la fusion, (ii) la substitution 'bind', (iii) le remplacement 'override'. La fusion est utilisée pour la composition de deux Themes de base. La substitution est surtout utilisée pour la composition d'un Theme de base avec un Theme d'aspect. La relation de remplacement est utilisée quand le comportement décrit par un Theme doit être remplacé par un nouveau comportement décrit par un autre Theme. Chaque relation de composition est associée à une stratégie d'intégration. Par exemple, dans le cas de la fusion, cette relation spécifie comment deux modèles ayant des concepts correspondants sont fusionnés.

Pour définir et formaliser la notion de composition, Theme\UML propose une extension du méta-modèle UML. En suivant le principe utilisé dans UML pour définir les éléments des modèles participant dans une relation (par exemple la relation de généralisation), Theme\UML définit le concept *d'élément composable* qui désigne tout élément participant à une relation de composition. Les éléments composables sont divisés en deux catégories : *primitifs* (attributs, opérations, etc.), et *composites* (classes, paquetages, etc.).

La relation de composition proposée dans [Clarke, 02] représente un nouveau type de relation, définie comme une sous-classe abstraite de la méta-classe '*Relationship*' du méta-modèle UML (Figure III.5). Deux éléments composables reliés par une relation de composition sont dits correspondants. Selon le type des éléments reliés, deux types de relations sont à distinguer : les relations de composition *primitives* reliant les éléments primitifs, et les relations de composition *composites* reliant les éléments composites. La méta-classe *Match* spécifie les critères de correspondance entre les éléments des modèles. Ces éléments sont intégrés selon la sémantique de la stratégie d'intégration associée à la relation de composition qui les relie.

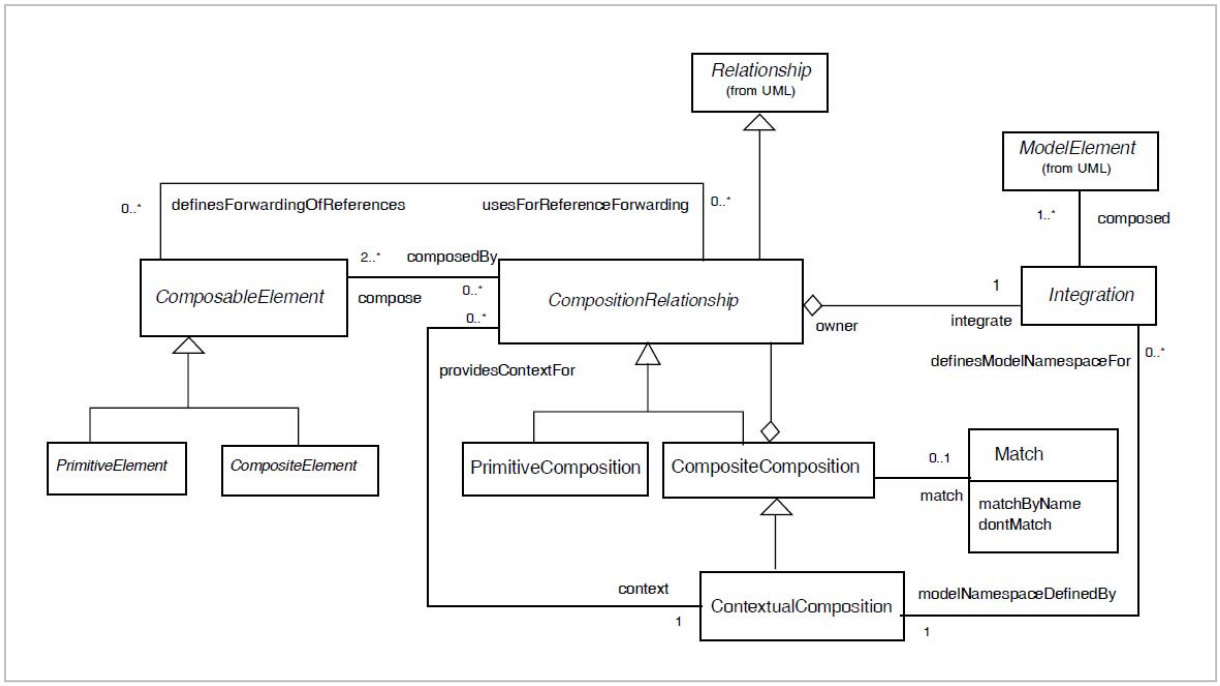


Figure III.5. Relation de composition dans Theme\UML [Clarke, 02]

III.3.2.2. Composition de modèles dans l'approche AAM

Les travaux décrits dans [France et al., 04a ; Reddy et al., 06] s'inscrivent dans le contexte de l'approche de modélisation par aspects AAM (*Aspect Architecture Modelling*). Cette approche permet de gérer la composition de modèles d'aspect modélisant les préoccupations transverses d'une application (persistance, sécurité, etc.), avec un modèle de base appelé *modèle primaire*. La technique de composition repose sur un algorithme de composition et un ensemble d'actions élémentaires appelées *directives de composition*. Ces directives permettent d'effectuer des opérations, soit sur les éléments des modèles, – comme la création/suppression d'un élément, la modification de la valeur d'une propriété, etc. –, soit sur les modèles d'aspects, en spécifiant par exemple l'ordre selon lequel deux modèles d'aspects sont composés. Un méta-modèle de composition est également proposé, il étend celui d'UML en ajoutant les spécifications des comportements de composition et inclut des méta-opérations qui implémentent la comparaison entre les éléments en se basant sur leurs signatures. L'utilisation de directives pour la résolution des conflits pouvant apparaître lors de la phase de composition s'avère intéressante car elle permet de réduire une activité qui était souvent à la charge du concepteur.

III.3.2.2.1. Aperçu de l'approche AAM

Dans l'approche AAM, le modèle d'architecture de l'application est défini par trois composantes : (i) le modèle primaire qui représente le cœur fonctionnel de l'application ; (ii) un ensemble de modèles d'aspect et les associations (*bindings*) utilisées pour instancier les aspects selon le contexte de l'application ; (iii) les directives de composition qui déterminent comment les modèles

d'aspect instanciés se composent avec le modèle de base pour produire le modèle composé de l'application.

Un modèle d'aspect est un patron caractérisant une famille de solutions logiques de préoccupations. Les patrons sont décrits par des paquetages paramétrés inspirés d'UML2 [OMG, 03a]. La notation employée est une adaptation d'un langage de description de patrons basé sur UML appelé RBML (*Rule-Based Metamodelling Language*) [France et al., 04b].

La composition d'un modèle d'aspect avec un modèle de base exige en premier lieu une étape d'instanciation du modèle d'aspect. Ceci est fait en substituant les paramètres définis dans l'aspect générique par les valeurs spécifiques de l'application. Le modèle d'aspect instancié est appelé modèle d'aspect spécifique à un contexte. La Figure III.6 présente le processus global de la composition dans AAM.

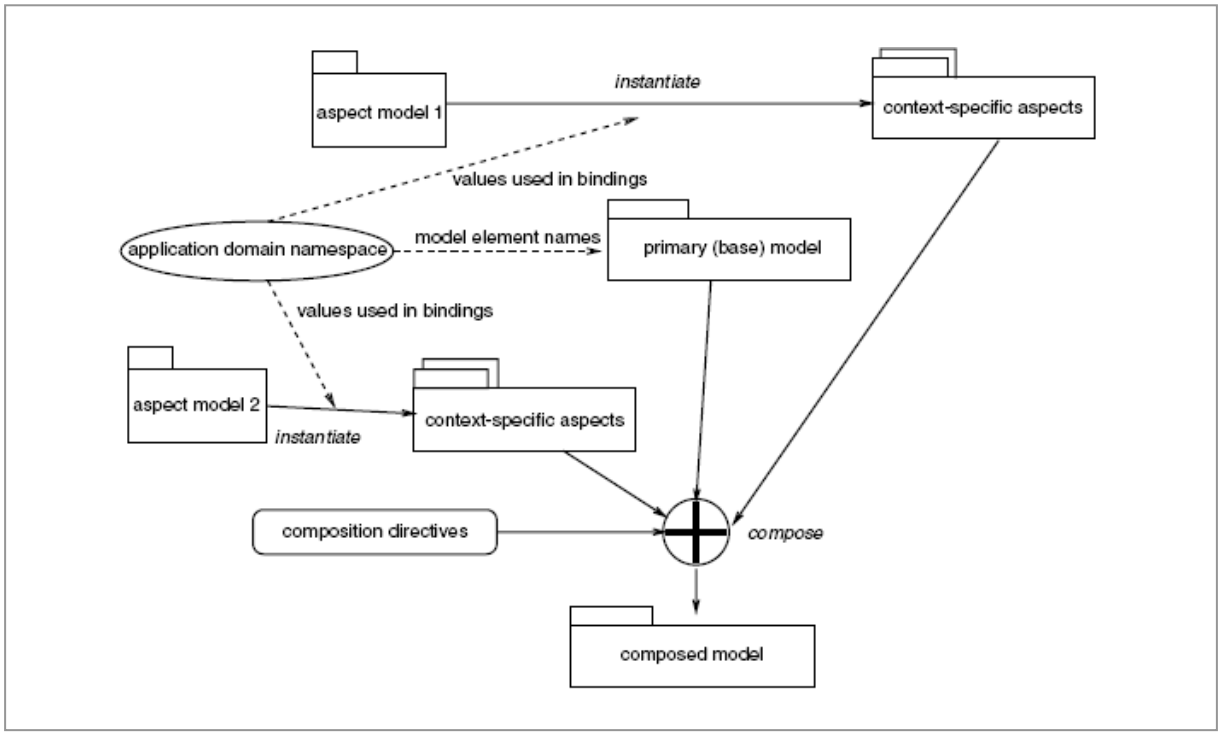


Figure III.6. Processus de composition de modèles dans l'approche AAM

III.3.2.2.2. Technique de composition utilisée

L'identification des éléments des modèles décrivant le même concept est basée sur la signature des éléments. Les éléments ayant la même signature sont fusionnés pour former un seul élément. Pour supporter l'alignement des éléments dans l'approche AAM, chaque type d'élément est associé à un type de signature [Reddy et al., 06] qui est un ensemble de propriétés syntaxiques.

La technique de composition de modèles basée sur la signature des éléments pourrait être entièrement automatisée, mais au risque de produire des résultats incohérents. Ceci peut mener par exemple, à apparier des éléments de modèles qui représentent des concepts différents. En outre, une telle technique ne permet pas de fusionner des éléments qui ont des signatures différentes tout en

représentant le même concept. À l'heure actuelle, l'approche AAM ne supporte pas la détection automatique des incohérences entre les modèles avant qu'ils soient composés. Par contre, cette approche propose la notion de directives de composition, et décrit comment ces directives peuvent être utilisées par les concepteurs pour assurer la compatibilité entre les modèles une fois les sources des incohérences identifiées.

L'utilisation de noms pour identifier les éléments correspondants, oblige les concepteurs à utiliser des noms cohérents dans leurs modèles. Ce n'est généralement pas le cas si les modèles à composer sont développés séparément par différentes équipes de concepteurs. Les directives de composition peuvent être employées dans ce but. Ainsi, les concepteurs peuvent spécifier des directives de composition pour modifier les modèles d'aspects et/ou le modèle de base, pour ajouter de nouveaux éléments au modèle composé, ou pour surcharger les règles prédéfinies de fusion.

Les directives de composition peuvent être classées selon le niveau de granularité des éléments auxquelles elles s'appliquent. Ainsi, on distingue les directives *d'éléments* des directives *de modèles*. Les directives d'éléments sont utilisées pour renommer, créer, supprimer, ajouter des éléments à un modèle. Les directives de modèles sont utilisées pour déterminer l'ordre selon lequel les modèles d'aspect sont composés avec le modèle de base, définissant ainsi une relation d'ordre de tissage entre les modèles d'aspect.

Le méta-modèle de composition présenté dans [France et al., 07] définit les constructions statiques et comportementales requises pour supporter la composition de modèles. Les propriétés comportementales sont spécifiées en termes d'opérations définies dans les méta-classes et par des descriptions textuelles associées à ces opérations. Ce méta-modèle décrit les extensions faites sur le méta-modèle d'UML pour supporter la fusion de modèles en utilisant les signatures et les règles de fusion et également pour supporter l'utilisation des directives de composition. Ce méta-modèle a été implémenté pour fournir un support automatisé de composition de modèles de classes en utilisant le langage Kermeta [Muller et al., 05].

III.3.2.3. Composition de modèles dans l'approche MATA

III.3.2.3.1. Aperçu de l'approche MATA

L'approche MATA (*Modeling Aspects using a Transformation Approach*) [Whittle et al., 07a] [Whittle et al., 07b] est une approche de composition de modèles d'aspect qui utilise les techniques de transformation de modèle. La procédure de composition est asymétrique, car elle fait la distinction entre un modèle de *base* et un modèle d'*aspect*. MATA définit un modèle d'aspect comme une combinaison de deux parties dépendantes : un patron et une spécification de composition. Le patron est utilisé pour détecter un emplacement dans le modèle de base où les spécifications de composition seront appliquées. Même si les modèles d'aspect et le modèle de base utilisent la même syntaxe concrète, ils sont différents à cause de la présence des variables de patron, et des annotations utilisées par la spécification de composition.

MATA définit trois types d'annotations représentés par les stéréotypes *create*, *delete* et *context*. Le stéréotype *create* est utilisé pour annoter les éléments qui vont être ajoutés dans le modèle de base, alors que les éléments marqués par le stéréotype *delete* vont être supprimés du modèle de base. Le stéréotype *context* est utilisé pour éviter d'appliquer un stéréotype à plusieurs éléments dans le cas où

un élément est annoté par un de ces stéréotypes et contient d'autres éléments. Le processus de composition avec MATA se fait en deux temps : d'abord un motif décrit par le patron de l'aspect est recherché dans le modèle de base, puis on procède à la modification de ce motif selon la spécification de composition.

L'approche MATA a été développée initialement dans le contexte de la modélisation orientée aspect. La technique de composition peut être généralisée à la composition de plusieurs modèles, en considérant une chaîne de transformation par application successive de modèles d'aspect. Bien que l'approche ne supporte a priori que la composition des diagrammes de classes, des diagrammes de séquence et d'états d'UML, elle peut être adaptée à d'autres modèles UML ou d'autres langages de modélisation décrits par un méta-modèle.

III.3.2.3.2. Principe de la composition

MATA considère la composition d'aspects comme un cas particulier de transformation de graphes. Cette transformation est définie par des règles de réécriture de graphe. Une règle prend en entrée un modèle de base M_b et un modèle d'aspect M_a et produit par fusion le modèle composé M_{ab}. Elle est de la forme : r : LHS→RHS. La partie gauche LHS (Left Hand Side) définit un patron qui spécifie un ensemble de points de coupure définissant des emplacements dans le modèle M_b où les nouveaux éléments doivent être ajoutés. La partie droite de la règle (Right Hand Side) spécifie les éléments et la manière dont ces éléments doivent être ajoutés au modèle M_b. La Figure III.7 décrit un exemple simple d'application d'une règle définie par un patron.

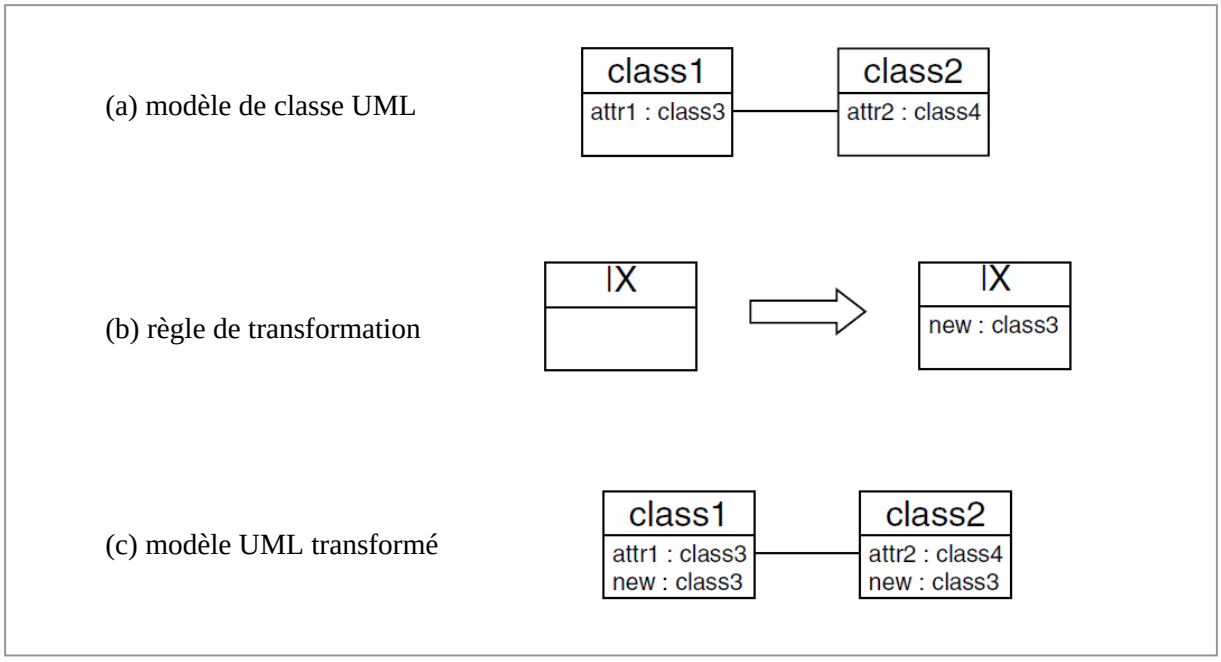


Figure III.7. Exemple de règle de transformation d'un modèle de classe UML [Whittle et al., 07b]

La Figure III.7-a représente le modèle de classe UML à transformer, la Figure III.7-b décrit la règle de transformation et la Figure III.7-c présente le résultat d'application de cette règle sur le modèle initial. La partie gauche de la règle décrit le patron sur lequel elle est appliquée ; ici, le patron

correspond à toute classe UML. La partie droite de la règle décrit les éléments qui doivent être ajoutés ou supprimés ; dans ce cas, l'attribut *new* est ajouté.

La particularité de cette approche est que les règles de transformation de graphe sont définies en utilisant la syntaxe concrète du langage de modélisation. Cette propriété distingue cette approche des approches de transformations les plus connues, par le fait que ces approches définissent la transformation au niveau du méta-modèle, en utilisant la syntaxe abstraite du langage de modélisation. Cela peut représenter un avantage certain pour le concepteur familier de la modélisation par aspect qui n'a pas une connaissance approfondie du méta-modèle UML, lorsqu'il doit décrire des transformations basées sur la syntaxe abstraite du langage de modélisation.

III.3.2.3.3. Exemple de composition de machines à états avec MATA

L'approche MATA offre un ensemble de patrons décrivant les règles de composition par type de diagramme UML à composer. La Figure III.8 donne des exemples de patrons appliqués dans le cas des diagrammes d'états.

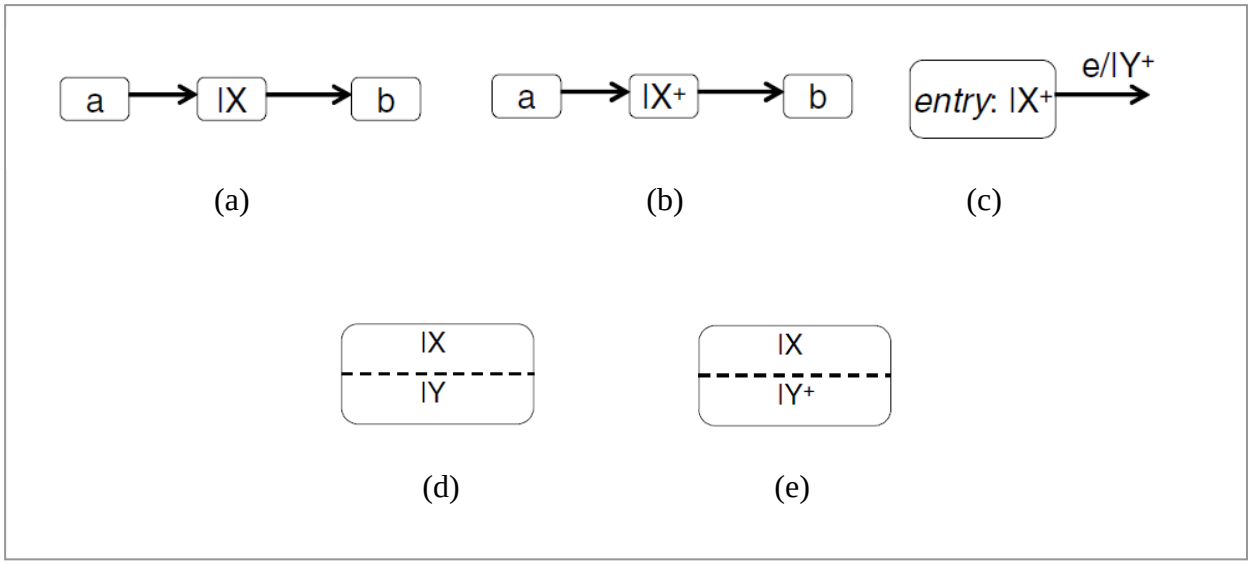


Figure III.8. Exemple de patrons de transformation de machines à états [Whittle et al,. 07-c]

La Figure III.8-a représente un patron décrivant la partie gauche d'une règle de transformation : ce patron correspond à tous les états a et b séparés par un état simple. La Figure III.8-b illustre un patron qui correspond à tous les états a et b séparés par un nombre quelconque d'états et de transitions. La Figure III.8-c décrit un patron qui correspond aux états ayant un ou plusieurs 'entry action' et ayant une transition sortante avec un événement 'e' et une ou plusieurs actions. La Figure III.8-d est un patron qui correspond aux états composés d'exactement deux régions orthogonales, tandis que la Figure III.8-e décrit le patron correspondant aux états composés d'au moins deux régions orthogonales.

L'exemple suivant illustre comment l'approche MATA peut être utilisée pour spécifier et composer les modèles d'aspect à travers une application de téléphonie mobile [Whittle et al., 07b]. Dans cet exemple, on se limite à la modélisation de trois cas d'utilisation : *réceptionner un appel*, *prendre*

un appel en messagerie et *notifier un appel en attente*. On considère le cas d'utilisation *réceptionner un appel* comme cas de base, car tous les téléphones mobiles disposent de cette fonctionnalité. Par contre, les deux autres cas d'utilisation seront considérés comme des aspects pour une meilleure séparation des préoccupations. Dans le processus de modélisation avec MATA, le cas d'utilisation de base est modélisé en UML, alors que les cas d'utilisation considérés comme des aspects sont modélisés selon le profil MATA. La Figure III.9 illustre la composition des machines à états issues des différents scénarios avec MATA.

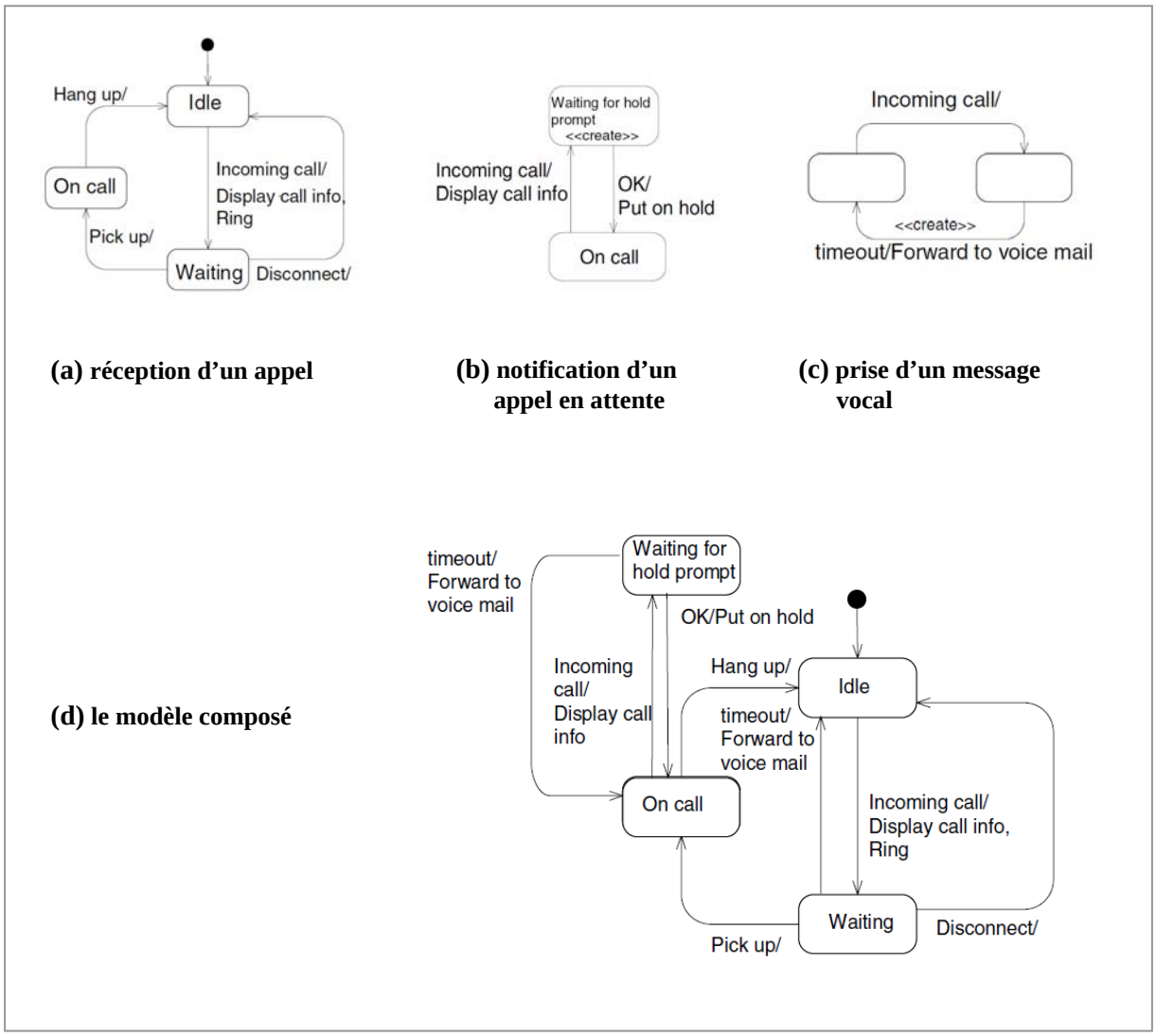


Figure III.9. Composition de machines à états avec l'approche MATA

La Figure III.9-a décrit le diagramme d'états du modèle de base 'réceptionner un appel'. Lors de la réception d'un appel, le téléphone informe l'utilisateur en affichant les informations de l'appelant (numéro, nom, etc.), et par l'envoi d'un signal sonore. L'utilisateur peut ensuite soit accepter l'appel (et raccrocher par la suite) soit le refuser. La Figure III.9-b décrit le diagramme d'états correspondant au scénario 'prendre un message vocal' modélisé par un modèle d'aspect selon l'approche MATA. Quand le téléphone sonne pendant une période de temps bien définie, l'appel est redirigé vers le système de messagerie vocale. La Figure III.9-c décrit un aspect dont la règle apparie les états qui ont une transition avec un événement nommé '*Incoming call*'. L'effet de l'aspect est d'ajouter une transition

supplémentaire pour décrire le comportement de prise d'un message vocal. La Figure III.9-d présente le modèle composé résultant de la composition du modèle de base et des deux modèles d'aspect.

III.3.3. Synthèse

L'objectif de cette thèse étant de développer une extension de VUML, nous avons sélectionné les travaux qui proposent des solutions basées sur l'extension du méta-modèle d'UML. Ces approches proposent de définir le comportement de la composition dans le méta-modèle définissant le langage de modélisation. En dépit des différences entre les sémantiques de composition de ces approches, le but de la composition est toujours le même : l'unification de concepts partagés par les modèles de conception.

L'approche Theme\UML propose une solution intéressante sur le plan méthodologique. En effet, la décomposition du modèle de conception en '*Themes*' facilite la compréhension du système, et permet de résoudre les problèmes de conception tels que la dispersion de la modélisation d'une exigence dans plusieurs éléments de modélisation, ainsi que l'enchevêtrement de plusieurs exigences dans une même unité de conception. Pour les outils support de cette approche on peut citer KerTheme [Jackson et al., 06].

L'approche AAM propose une solution pour résoudre les problèmes de composition en utilisant un ensemble de directives de composition. Le point fort de cette approche est de proposer des mécanismes de composition bien définis et implémentés avec le langage Kermeta [France et al., 07], et mis en œuvre dans l'outil Kompose [Fleurey, 07]. Notons toutefois que cette approche couvre essentiellement les modèles de classes d'UML, et ne propose pas d'analyse des modèles composés. L'identification des problèmes de composition reste à la charge du concepteur qui doit spécifier les directives de composition appropriées à appliquer.

L'approche MATA présente une technique de composition de modèles d'aspect basée sur le langage UML et le profil MATA. Dans cette approche, la composition est vue comme un cas particulier de transformation de modèle, ce qui permet d'assoir la composition d'aspects sur une base formelle en utilisant la théorie des graphes. La technique supporte les modèles structurels d'UML (diagrammes de classes), et les modèles de comportement (diagrammes de séquences et machines à états). A la différence de certaines approches de modélisation orientées aspect [Cottenier et al., 07], le mécanisme de composition dans MATA ne se base pas sur l'identification des points de jonctions, car ceci est réalisé par le modèle d'aspect lui-même. MATA se base sur la syntaxe concrète du langage de modélisation, ce qui facilite la tâche du modélisateur, mais ne favorise pas la réutilisation des transformations qui sont spécifiées explicitement dans les modèles d'aspect.

Nous résumons les techniques de décomposition et de composition de modèles adoptés par les approches étudiées dans le tableau suivant :

Approche	Technique de décomposition utilisée	Technique de composition utilisée	Outils support
Theme\UML	Décomposition en thèmes : thème de base et thèmes d'aspect. - un Theme est représenté par un paquetage paramétré. Chaque paramètre correspond à une classe et à l'ensemble des méthodes sur lesquelles la fonctionnalité doit être tissée.	Trois types de composition : - fusion : composition de deux Themes de base, - substitution 'bind' : composition d'un thème de base et d'un thème d'aspect, - override : remplacement d'un thème par un autre.	- Extension du méta-modèle UML. - L'outil KerTheme
AAM	Décomposition en : - un modèle primaire (cœur fonctionnel de l'application), - des modèles d'aspect et les associations (bindings) utilisées pour instancier les aspects selon le contexte de l'application.	- la composition repose sur un algorithme (basé sur la signature des éléments) et un ensemble de directives de composition. - les directives de composition déterminent comment les modèles d'aspect instanciés se composent avec le modèle de base.	- Un méta-modèle de composition étendant celui d'UML. - L'outil Kompose, basé sur Kermeta
MATA	Décomposition en : - un modèle de base - un modèle d'aspect	- composition par transformation de modèle. - la transformation est basée sur un patron utilisé pour détecter un emplacement dans le modèle de base où les spécifications de composition seront appliquées.	- Profil MATA (profil UML)

III.4. VUML

Le contexte général de notre travail de thèse est le profil VUML [Nassar, 05 ; Anwar, 09]. Dans cette section, nous présentons d'abord les principes de base de VUML, puis nous décrivons la notion de classe multivue ainsi que les mécanismes associés. Nous illustrons les différents concepts par des extraits de l'étude de cas "Gestion d'une agence de réparation de voitures" qui sert de référence pour les illustrations de cette thèse.

III.4.1. Origine et principe

Le langage VUML (*View based UML*) est un profil UML basé sur l'approche de modélisation par points de vue. Ce profil offre un formalisme étendant celui d'UML et une démarche inspirée de celle de la méthode VBOOM (*View Based Object Oriented Methodology*) [Kriouile 95]. Outre le fait de s'appuyer sur un formalisme non standard inspiré du langage Eiffel, VBOOM souffrait d'un certain nombre de limitations dont l'implantation des vues par de l'héritage multiple et le fort non déterminisme dans l'identification des vues.

VUML, tout en visant des objectifs similaires à ceux de VBOOM, s'appuie explicitement sur le standard UML, et introduit un ensemble de concepts et de mécanismes pour : (i) gérer les droits d'accès aux classes multivue, (ii) spécialiser une classe multivue, (iii) spécifier les dépendances entre les

vues, (iv) assurer la cohérence du modèle en cas de mises à jour, et (v) administrer les vues à l'exécution.

Informellement, les concepts clé de VUML sont définis comme suit :

- **Acteur :** humain ou entité logique qui interagit avec le système.
- **Point de vue** : vision d'un acteur sur le système (ou sur une partie de ce système). Un point de vue unique est associé à un acteur.
- **Vue :** entité de modélisation (statique). Elle correspond à l'application d'un point de vue sur une entité donnée (classe, et par généralisation le système entier). Par simplification de langage, nous dirons qu'une vue est associée à un acteur en considérant comme implicite l'entité sur laquelle le point de vue de l'acteur s'applique.

III.4.2. Concepts de base

III.4.2.1. Classe multivue

Une **classe multivue** est définie comme une unité d'abstraction et d'encapsulation permettant de stocker et de restituer l'information en fonction du profil de l'utilisateur. Une telle classe est composée d'une base (stéréotype « base ») qui a le même nom que la classe UML correspondante, et de vues (stéréotype « view ») qui sont reliées à la base via une relation de dépendance appelée « viewExtension ». La relation viewExtension n'est pas une relation d'héritage : les vues dépendent de la base au sens où les attributs et les méthodes de la base sont implicitement partagés par les vues de la classe multivue. Une caractéristique d'une vue peut redéfinir une caractéristique de la base. La Figure III.10 ci-dessous illustre la structure statique d'une classe multivue.

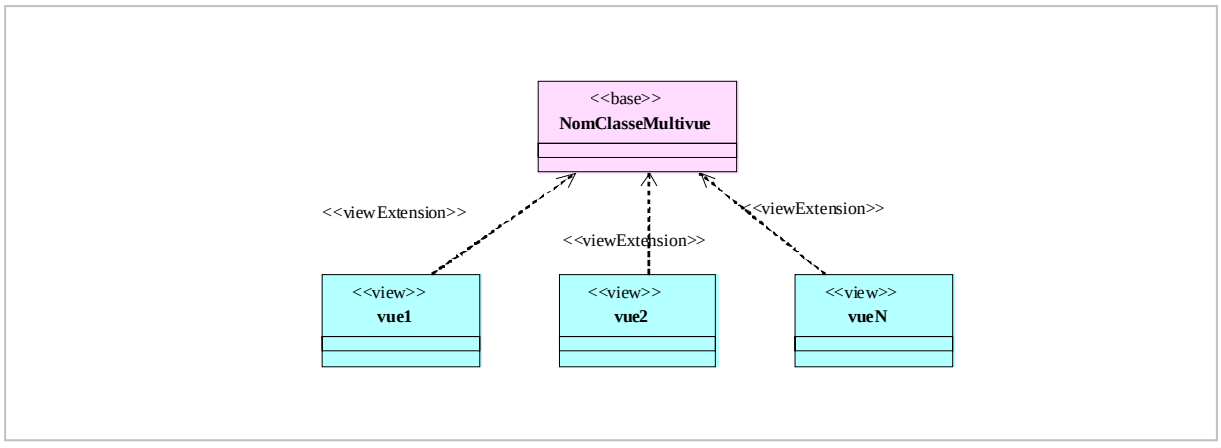


Figure III.10. Structure statique d'une classe multivue [Nassar, 05]

La Figure III.11 illustre cette structure d'une classe multivue à travers l'exemple simplifié de la classe multivue *Voiture*, constituée d'une base et de quatre vues correspondant aux points de vue du mécanicien, du chef d'agence, du client et du responsable d'atelier.

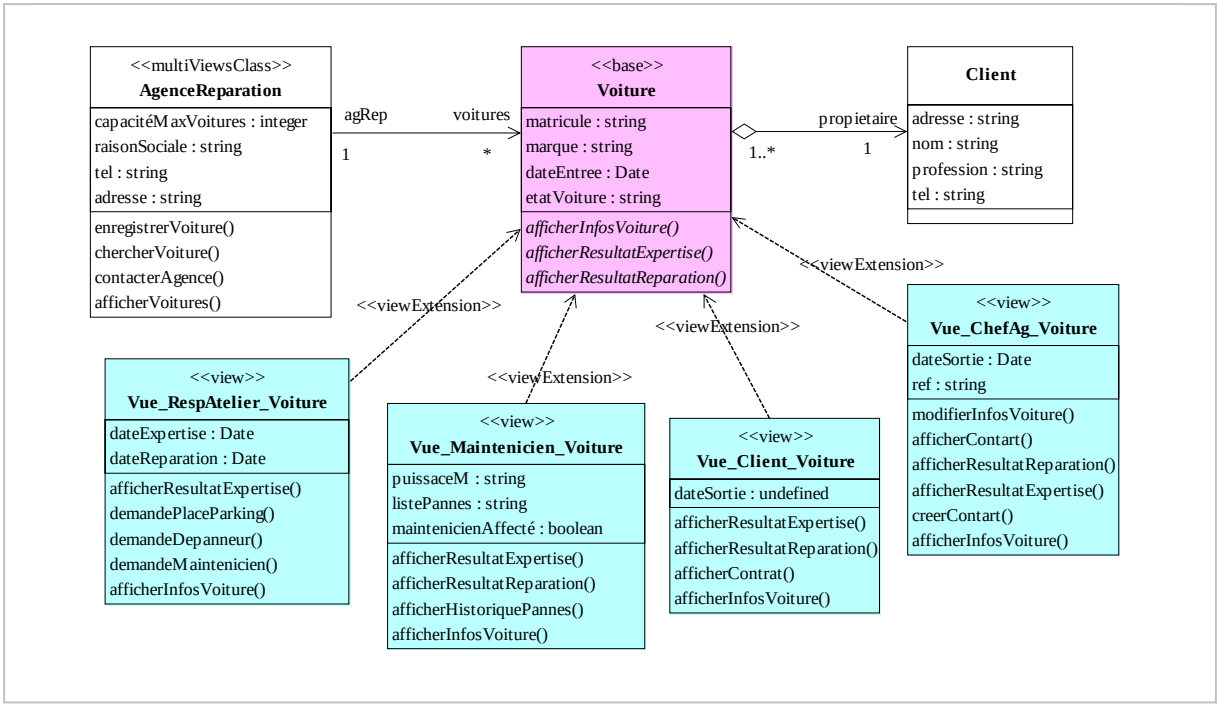


Figure III.11. Exemple simplifié de la classe multivue Voiture

La base contient les caractéristiques (attributs et méthodes) considérées comme irréductibles, c'est-à-dire indépendantes d'un point de vue donné. La classe *Vue_ChefAgence_Voiture* décrit la spécificité d'une voiture selon le point de vue d'un chef d'agence, tandis que la classe *Vue_Maintenicien_Voiture* décrit la spécificité d'une voiture selon le point de vue d'un maintenicien. On peut remarquer que la méthode *afficherInfos()* de la base est redéfinie dans les quatre vues. Toutes les classes d'un système sont potentiellement multivue car on peut toujours ajouter un point de vue non identifié initialement au modèle.

III.4.2.2. Dépendances entre les vues

Les vues d'une classe multivue peuvent naturellement être dépendantes. Autrement dit, les modifications de valeurs d'attributs d'une vue peuvent avoir des répercussions sur les valeurs d'attributs d'autres vues. Il est donc nécessaire de maintenir la cohérence interne d'une telle classe. La gestion des répercussions fait habituellement partie de la phase d'implémentation au sens où du code doit être introduit pour faire les mises à jour nécessaires. Cependant, il est préférable que ces dépendances entre les vues soient explicitées le plus tôt possible dans le développement, c'est-à-dire durant la phase de conception ; pour cela une relation de dépendance stéréotypée par « viewDependency » a été introduite en utilisant les notes d'UML ou le langage OCL (*Object Constraint Language*) [OMG-OCL] pour spécifier les contraintes.

La Figure III.12 présente un exemple abstrait d'une dépendance entre deux vues d'une classe multivue. La dépendance « viewDependency » entre les vues vue1 et vue2 indique que des données de la vue vue1 (source de la dépendance) dépendent de certaines données de la vue vue2 (cible de la dépendance). La relation entre ces données doit être décrite en OCL sur la note associée à cette dépendance. Notons qu'il existe plusieurs types de dépendances entre les vues : dépendance

d'inclusion de données, dépendance d'égalité de données et dépendance fonctionnelle [Nassar et al., 03].

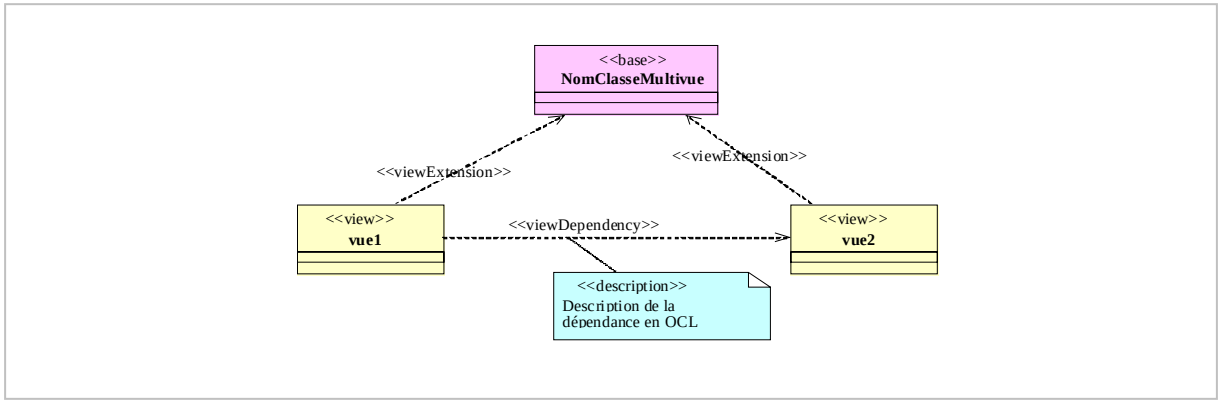


Figure III.12. Illustration abstraite d'une dépendance entre deux vues

III.4.3. Méta-modèle de VUML

La Figure III.13 donne un aperçu global du méta-modèle VUML, les éléments ajoutés à UML étant marqués en couleur foncée.

Nous ne donnons ici que la sémantique informelle associée à VUML. Cette sémantique informelle a été traduite en règles de bonne modélisation ou WFR (*Well-Formedness Rules*) et présentée dans [Nassar, 05]. La sémantique informelle associée aux éléments du méta-modèle VUML est la suivante :

- **Base** : élément spécialisant la méta-classe *Class* et qui décrit les caractéristiques structurelles et comportementales communes aux acteurs du système. L'élément *Base* peut être associé à des relations de dépendance *ViewExtension* qui permettent de rattacher des vues à cette base.

- **View** : élément permettant de modéliser les caractéristiques structurelles et comportementales spécifiques à un acteur donné. Il est relié avec la base via la relation *ViewExtension*. Il peut aussi être source ou cible d'une ou plusieurs relations *ViewDependency*.

- **MultiViewsClass** : élément spécialisant la méta-classe *Classifier*. Il est composé d'une Base et d'une liste de *View* reliés à la base via des *ViewExtension*.

- **ViewExtension** : élément spécialisant la méta-classe *Dependency*. C'est une dépendance ayant comme source une vue et comme cible une base. Les vues dépendent de la base au sens où les attributs et les méthodes de la base sont implicitement partagés par les vues de la classe multivue.

- **ViewDependency** : élément permettant la modélisation des relations de dépendance entre les vues. Chaque *ViewDependency* peut être associée à une ou plusieurs contraintes qui peuvent être exprimées soit en langage naturel, soit en langage formel tel qu'OCL.

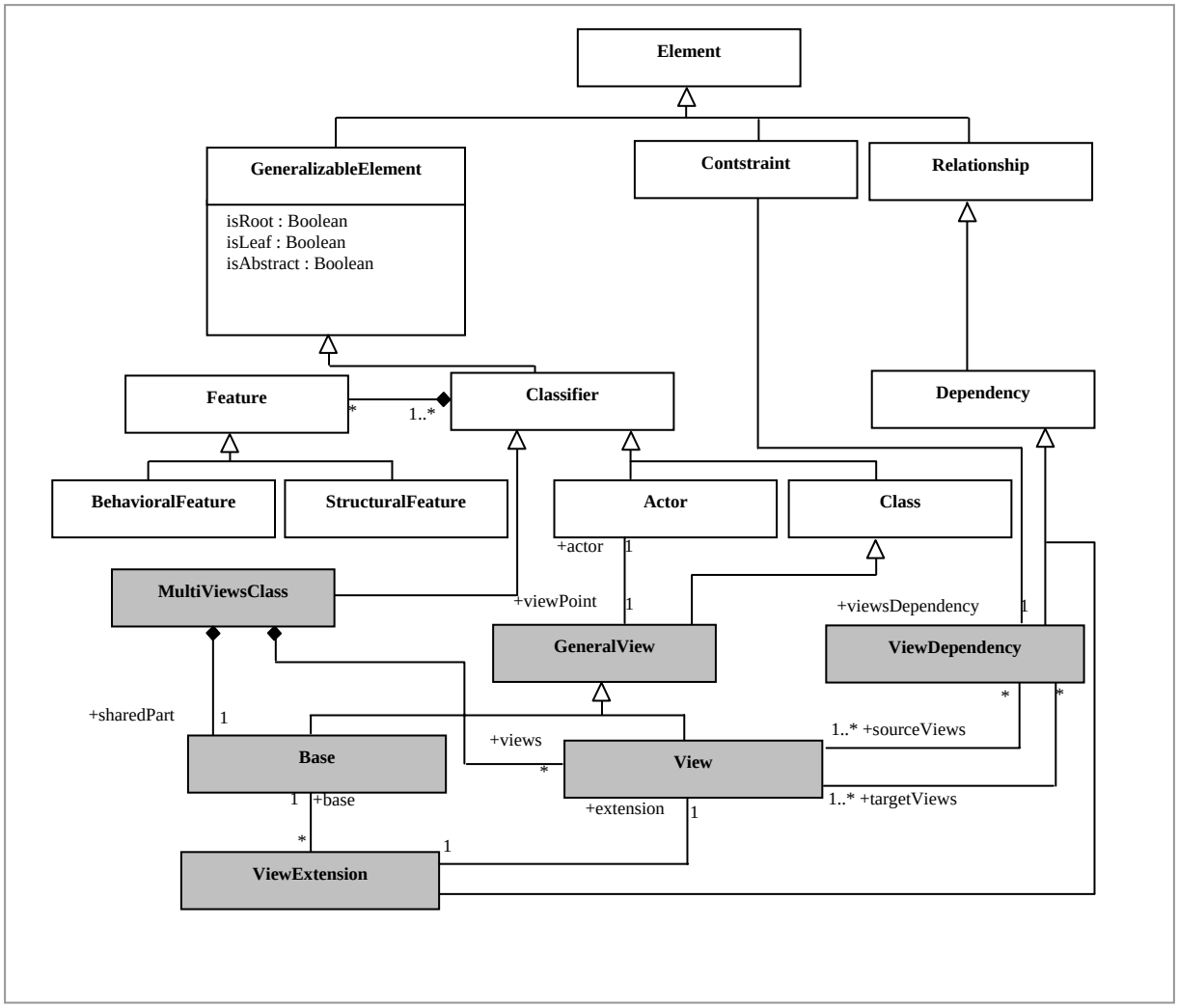


Figure III.13. Fragment du méta-modèle associé au profil VUML [Nassar, 05]

III.4.4. Démarche VUML appliquée à un cas d'étude

Après un rappel de la démarche VUML [Nassar, 05], nous illustrons ses étapes principales sur un cas d'étude.

III.4.4.1. Principe de la démarche VUML

Comme nous le montrons dans la Figure III.14, la démarche VUML comporte trois phases de développement principales : une phase centralisée de modélisation des exigences, une phase décentralisée de modélisation du système suivant chaque point de vue (en utilisant les différents diagrammes UML), et enfin une phase centralisée de fusion et de modélisation produisant le diagramme de classes VUML.

Dans un premier temps, la modélisation des besoins est réalisée sous forme de cas d'utilisation ; puis, pour chaque acteur identifié (donc pour chaque point de vue) sont spécifiés les scénarios ainsi

que les diagrammes de classes associés ; la fusion VUML permet d'identifier les classes multivue et de réaliser un diagramme de classes global VUML.

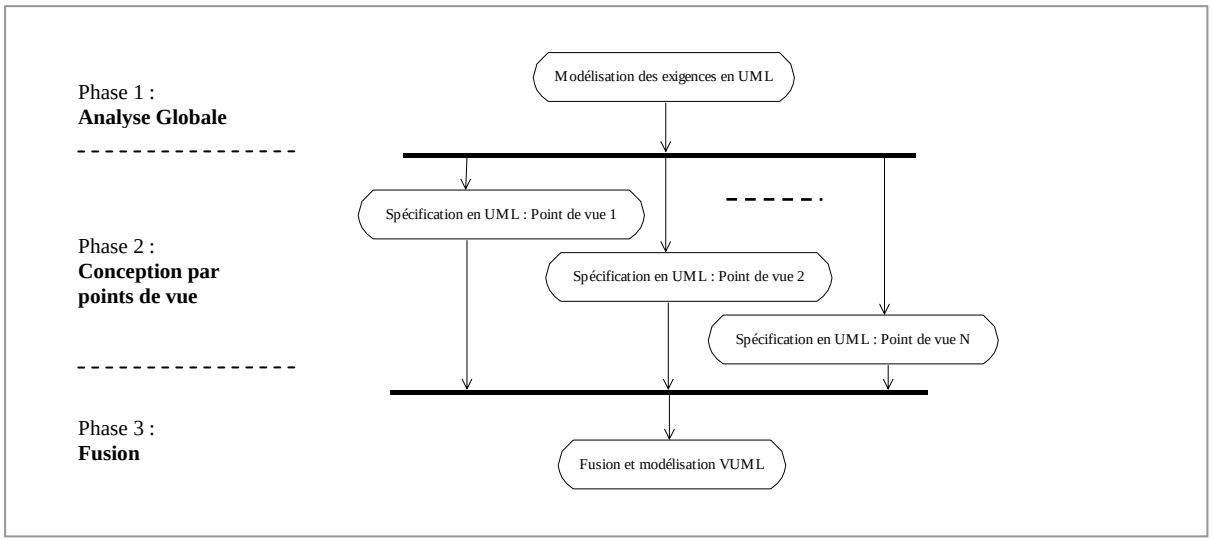


Figure III.14. Vue générale de la démarche VUML

III.4.4.2. Illustration de la démarche VUML sur un cas d'étude

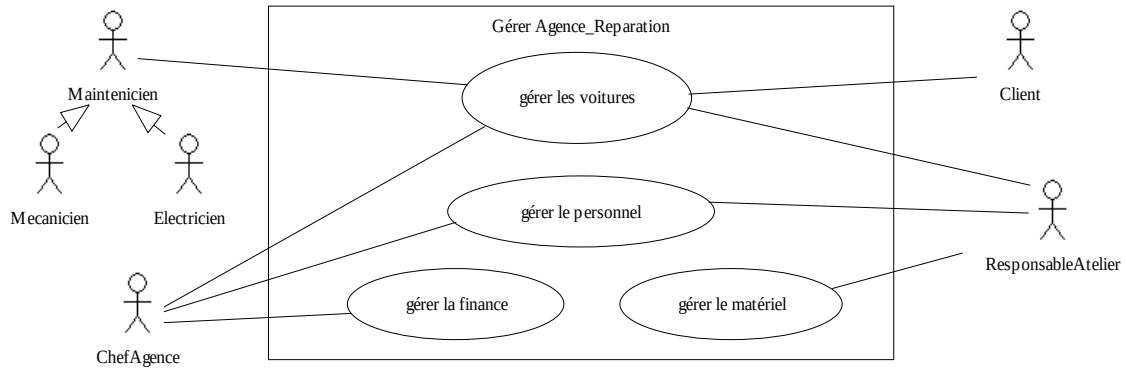
Nous illustrons dans les sous-sections suivantes les trois phases de la démarche VUML en présentant les étapes principales constituant chaque phase. Nous prenons comme support le cas d'étude que nous avons adopté pour illustrer les contributions de cette thèse. Ce cas d'étude est détaillé dans le chapitre V. Il s'agit du système de gestion d'une agence de réparation de voitures. C'est un système d'information complexe, impliquant plusieurs acteurs, et se prêtant donc bien à une conception par points de vue.

III.4.4.2.1. Analyse globale

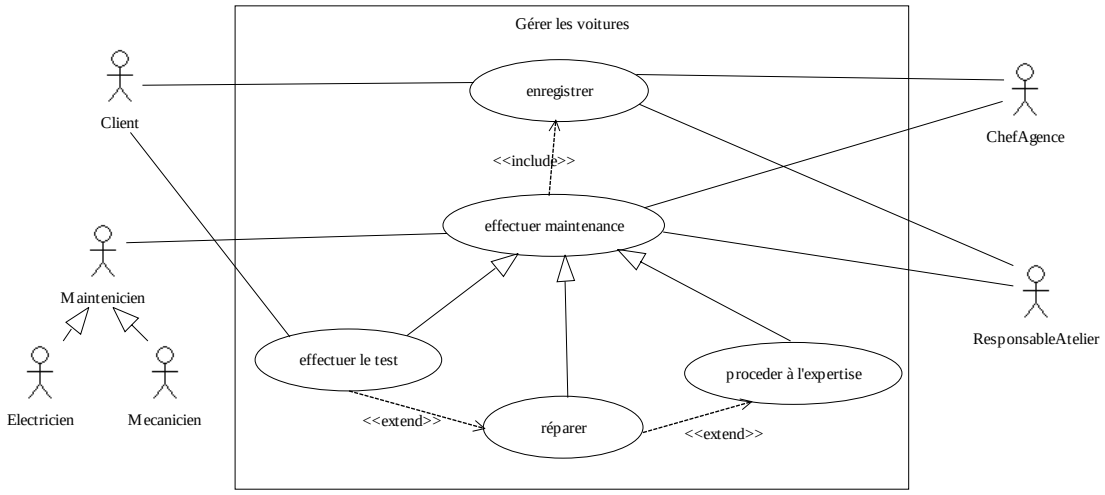
La première phase de la démarche est l'analyse globale. C'est une phase centralisée de modélisation des exigences. L'objectif est d'identifier les besoins des différents acteurs du système, et de les structurer en unités fonctionnelles sous forme de cas d'utilisation. La Figure III.15-a donne un aperçu général des fonctionnalités que le système doit assurer, à savoir : (1) la gestion des voitures, (2) la gestion du personnel, (3) la gestion du matériel, (4) la gestion financière et (5) la supervision de l'agence. Chacune de ces fonctionnalités est décomposée en unités fonctionnelles plus fines montrant la responsabilité affectée à chaque acteur. Nous avons limité l'étude aux acteurs suivants : le client, le chef d'agence, le responsable d'atelier, et le maintenicien.

A titre illustratif, nous ne développons ici que la fonctionnalité "gérer les voitures" (cf. Figure III.15-b). Gérer les voitures de l'agence revient à gérer leur enregistrement, les expertises et les réparations apportées à chacune d'elles, et les tests de vérification finaux. Un client intervient dans le cas d'utilisation "enregistrer" par la communication des informations concernant sa voiture, il participe avec le chef d'agence à la réalisation des contrats d'expertise et de réparation, et valide la réparation en effectuant le test final du bon fonctionnement de sa voiture. Les mainteniciens

interviennent dans les phases techniques telles que l'expertise, la réparation et les tests. Le responsable atelier participe quant à lui à la gestion des tâches de maintenance à effectuer sur la voiture.



(a) : Cas d'utilisation généraux de l'application "Gestion Agence Réparation"

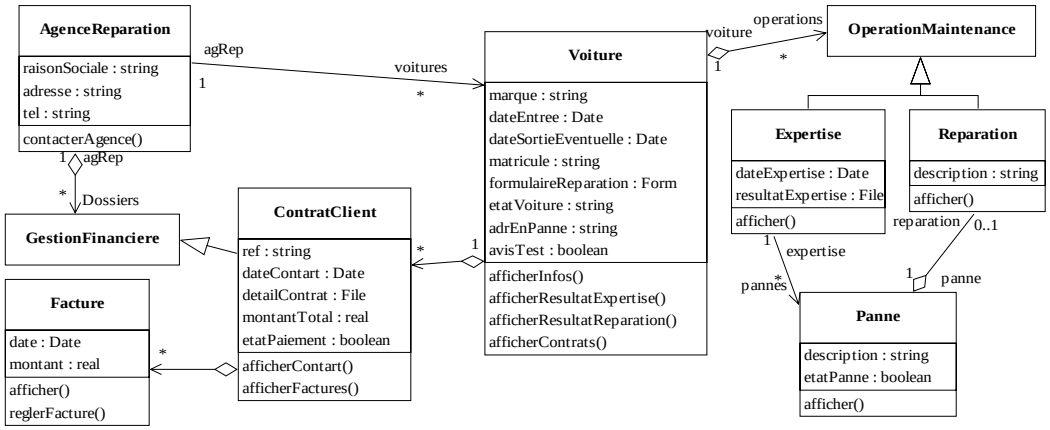


(b) : Cas d'utilisation de "Gérer les voitures"

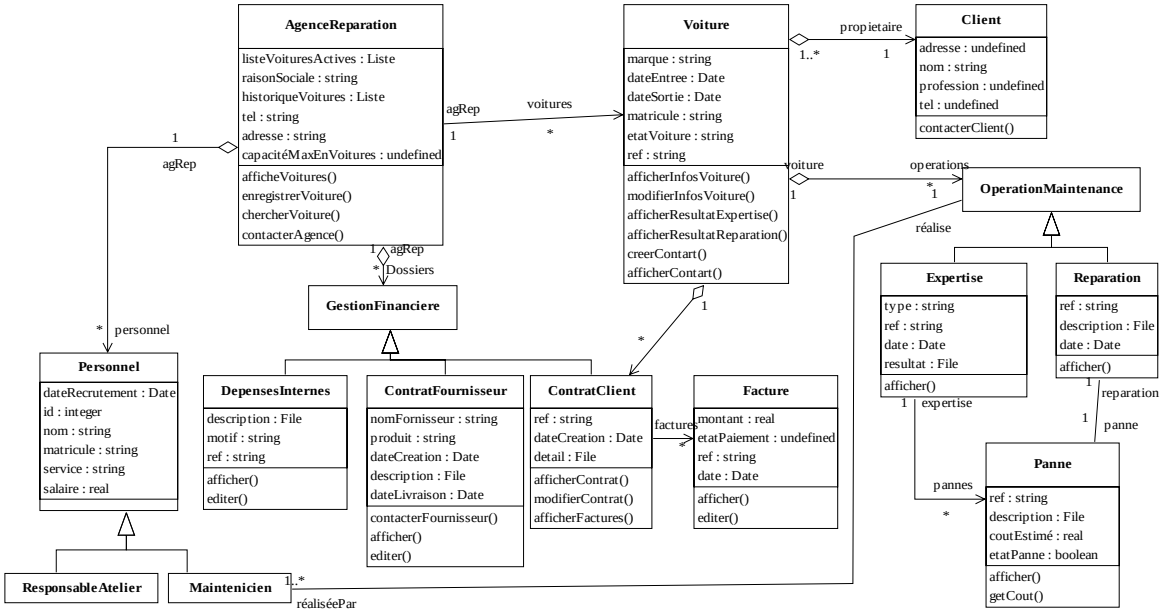
Figure III.15. Exemples de cas d'utilisation du système "Gestion d'une agence de réparation de voitures"

III.4.4.2.2. Conception par points de vue

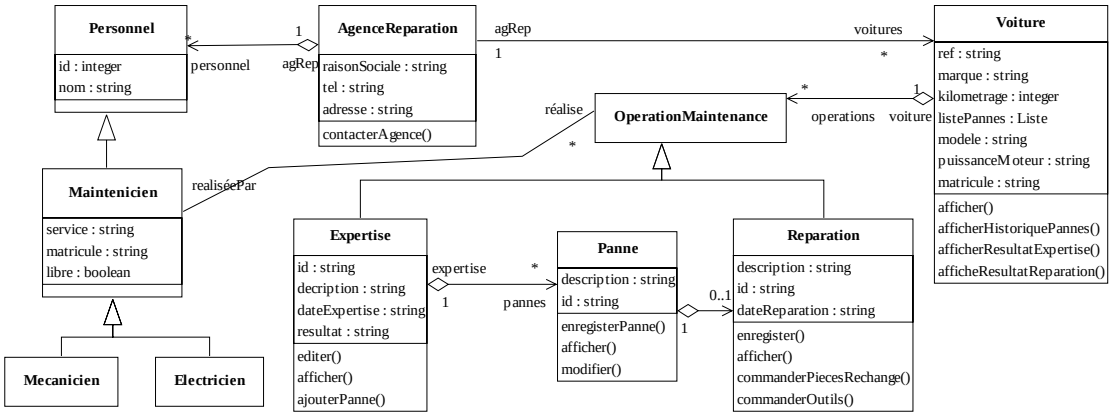
C'est une phase de conception décentralisée, au cours de laquelle plusieurs équipes de concepteurs peuvent travailler séparément pour réaliser des modèles de conception par points de vue. Pour chaque acteur, – donc pour chaque point de vue –, nous reprenons les cas d'utilisation auxquels il participe. Nous établissons les diagrammes de séquence correspondant à ces cas d'utilisation. Nous pouvons élaborer ensuite le diagramme de classe détaillé pour le point de vue considéré en rassemblant les informations déduites des diagrammes de séquence développés. Le résultat final de cette phase est un ensemble de modèles dont chacun représente les spécificités d'un point de vue donné. La Figure III.16 illustre les diagrammes de classes selon les points de vue *Client*, *ChefAgence* et *Mecanicien*.



(a) : Extrait du diagramme de classes UML – point de vue *Client*



(b) : Extrait du diagramme de classes UML – point de vue *ChefAgence*



(c) : Extrait du diagramme de classes UML – point de vue *Mecanicien*

Figure III.16. Extraits des diagrammes de classes UML résultant de la 2ème phase

III.4.4.2.3. Fusion

La phase de fusion est une étape clé dans la démarche associée à VUML [Anwar, 09]. Cette phase a pour but de produire le modèle VUML résultant de la fusion des modèles par point de vue élaborés précédemment. Dans le cas de notre exemple, le résultat final de la fusion est représenté par le diagramme de classes VUML de la Figure III.17. Signalons que nous pouvons présenter les classes multivue soit sous forme éclatée en faisant apparaître toutes les vues, soit sous forme "iconifiée" en spécifiant le stéréotype *« MultiViewsClass »*. Les classes multivue de cette application sont : *Voiture*, *Expertise*, *Reparation*, *Panne*, *ContratClient*, *Personnel* et *AgenceReparation*.

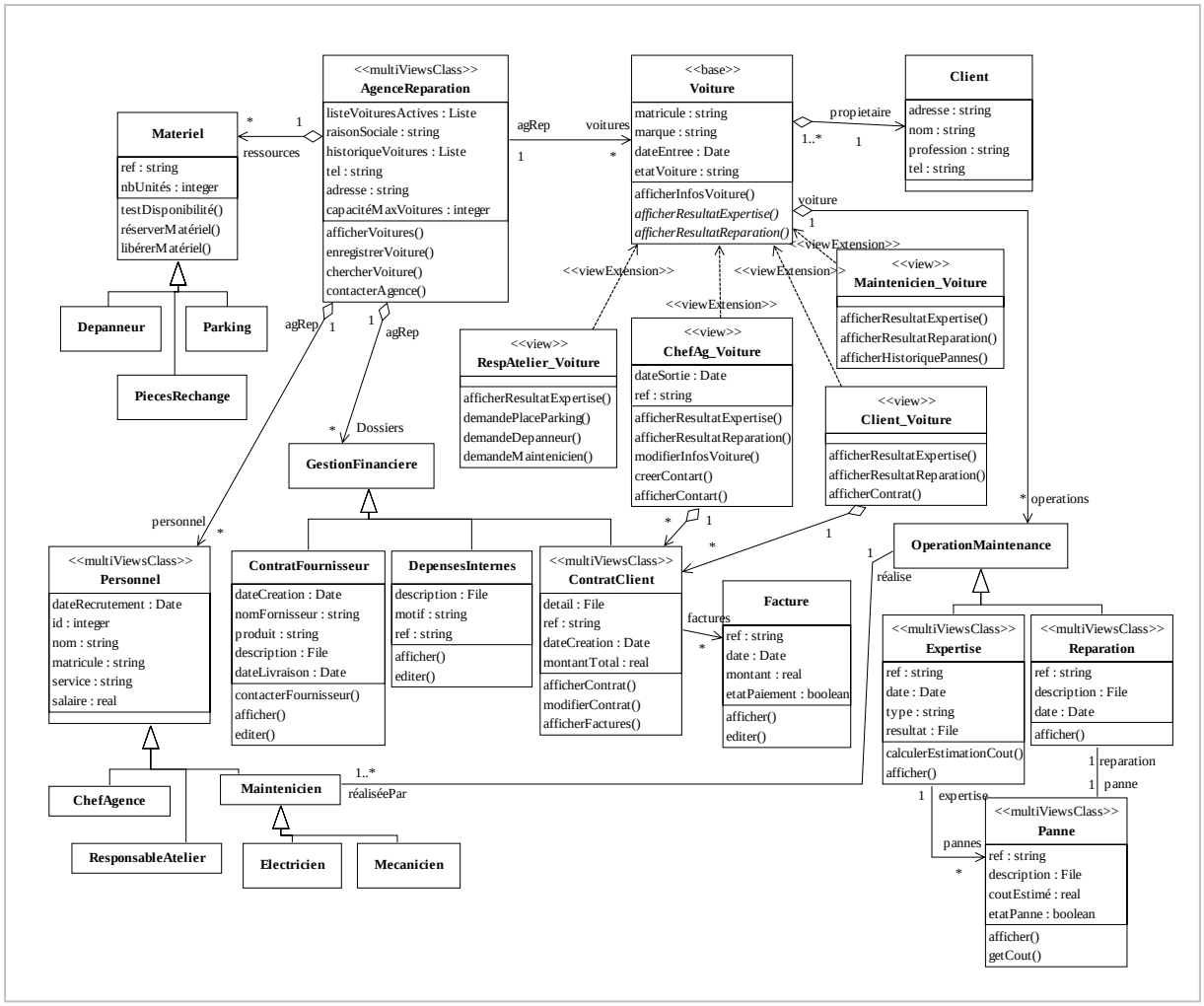


Figure III.17. Extrait du diagramme VUML obtenu par fusion des modèles partiels

A titre d'exemple, la classe multivue Voiture est composée d'une base partagée (stéréotype « base ») qui regroupe les attributs/méthodes accessibles par tous les acteurs, tels que *matricule*, *date Entree* et la méthode *afficherInfosVoiture()*, et de vues spécifiques aux acteurs (stéréotype « view »). Par exemple, l'attribut *dateSortie* et *modifierInfosVoiture()* ne sont accessibles que par l'acteur *ChefAgence*.

III.4.5. Synthèse sur VUML

Etant basé sur une approche de modélisation par points de vue, le langage VUML bénéficie des avantages de la technique de séparation des préoccupations, ce qui est utile dans le cas des systèmes complexes. De plus, par sa démarche centrée acteur, il offre un moyen pertinent de gestion des droits accès. On peut résumer l'objectif du profil VUML comme suit :

- *Il vise l'indépendance dans le développement des modèles par points de vue.* Dans la phase de conception d'un point de vue, on ne s'intéresse qu'aux préoccupations liées à l'acteur correspondant. VUML propose toutefois une phase préalable d'analyse globale avant d'entamer le développement parallèle des vues. Ceci permet d'harmoniser le vocabulaire et de définir un noyau de glossaire commun.
- *Il organise le partage des données entre les différentes vues.* Dans la phase de composition des modèles partiels, les données sont structurées en données partagées et données spécifiques à chaque acteur. Par le concept de classe multivue et par l'association viewExtension, VUML permet une structuration du partage des données dans une application.
- *Il traite la cohérence du modèle VUML final.* La phase de composition structurelle est soumise à des règles de correspondance, de fusion et de translation [Anwar et al., 08a] pour traiter la cohérence des modèles générés après la phase de fusion.
- *Il fait apparaitre les droits d'accès au niveau de la conception.* Contrairement aux approches classiques, où le contrôle d'accès aux informations est programmé dans les méthodes des classes, VUML les prend en compte à un niveau d'abstraction plus élevé, c'est-à-dire lors de la phase d'analyse/conception.
- *Il favorise la réutilisation et l'évolution.* La modularité obtenue par une séparation des préoccupations par vue, permet la modification des vues et rend possible à tout moment l'ajout d'un nouveau point de vue. La prise en compte d'un nouveau point de vue suit la même démarche de développement que pour les autres points de vue.

Procéder à une modélisation par partie du système avec VUML offre des avantages surtout dans le cas des systèmes complexes, mais cette démarche souffre principalement des deux limites suivantes :

(i) Le développement des vues doit tenir compte des dépendances entre les vues. En fait, le couplage entre les vues est important car la modélisation du point de vue courant nécessite des informations extérieures à ce dernier. Cette situation peut rendre la démarche de conception par points de vue difficile à mettre en œuvre, et peut nécessiter une coordination des développements des différentes vues afin de partager les informations nécessaires à l'intégration des vues. L'approche que nous proposons dans le chapitre IV minimise l'impact de la dépendance entre les vues en proposant des mécanismes d'intégration de vues spécifiques.

(ii) La fusion de modèles est un problème très complexe. Elle est réalisée jusqu'à maintenant d'une manière semi-automatique et nécessite l'intervention humaine malgré les efforts faits pour automatiser certaines tâches [Anwar, 09].

III.5. Problématique de spécification et de composition du comportement dans VUML

Les travaux réalisés sur le profil VUML ont été focalisés jusqu'à présent sur l'aspect structurel de la modélisation, mais sans prendre en compte l'aspect comportemental de la modélisation multivue. En fait, les travaux réalisés ont traité la structuration statique des applications VUML, telle que le partage des données et la composition statique des vues, sans traiter la manière dont ces vues vont réagir, ni comment les synchroniser afin de représenter le comportement des objets multivue (instances de classes multivue).

Or la modélisation comportementale est une étape importante dans la démarche de conception d'un système complexe, surtout dans le contexte de l'ingénierie dirigée par les modèles, où l'objectif est d'arriver à une automatisation des phases post-conception (codage, intégration, validation, etc.), automatisation qui doit être fondée sur un modèle de conception le plus complet possible. La modélisation comportementale en UML peut se faire à plusieurs niveaux d'abstraction, en partant des modèles d'ensemble comme les modèles d'interaction et d'activités qui représentent les interactions et l'enchaînement des activités entre les différents objets ou les composants du système, en allant jusqu'à la description fine du comportement des objets ou des composants par des machines à états. Les modèles d'ensemble, tels que le diagramme de séquence, permettent par définition la description d'un comportement selon un point de vue ou une combinaison de plusieurs points de vue.

Dans ce travail de thèse, nous nous concentrons sur la description du comportement individuel des objets multivue par des machines à états qui, elles, nécessitent des adaptations au niveau des concepts de modélisation UML. En fait, un objet multivue – instance d'une classe multivue – est un objet composé d'un ensemble de sous-objets, chacun encapsulant les données spécifiques à un acteur. Cependant, aucune étude n'a été faite sur la manière dont ces sous-objets communiquent, ni comment ils collaborent pour réaliser un besoin comportemental donné. Notre principal objectif dans cette thèse est de répondre à cette problématique de spécification comportementale dans un cadre orienté points de vue.

III.6. Conclusion

Dans la première partie de ce chapitre, nous avons passé en revue les principales approches de modélisation basées sur le principe de multi-modélisation. Ces approches préconisent dans une première étape le développement de modèles partiels, avec un objectif de réduction de la complexité de l'application, puis une composition de ces modèles partiels pour constituer le modèle final de l'application.

Dans la deuxième partie de ce chapitre nous avons présenté les travaux de recherche qui s'intéressent à la problématique de composition de modèles. Dans un premier temps, nous avons présenté les efforts de l'OMG concernant la composition de modèles dans UML. Dans un deuxième temps, nous avons donné un aperçu des approches de composition du comportement, notamment celles à base d'aspects qui rencontrent des problèmes de composition similaires aux nôtres. Notre objectif étant de développer une extension de VUML, nous avons sélectionné les travaux qui proposent des solutions basées sur l'extension du méta-modèle d'UML. L'étude que nous avons menée montre que jusqu'à présent il n'y a pas de démarche complète et automatisable pour la composition de modèles et encore moins de standard relatif à cette problématique.

Dans la troisième partie de ce chapitre nous avons présenté l'approche VUML qui est le profil sur lequel nous nous sommes appuyés pour ce travail de thèse. VUML a pour objectif d'intégrer les concepts de vue et de point de vue dans l'analyse/conception d'un système. Tout d'abord nous avons introduit les définitions concernant les notions de vue et de point de vue, puis la notion de classe multivue, qui est l'élément clé de cette approche. Nous avons ensuite présenté la démarche associée à VUML en l'illustrant à travers un cas d'étude. Nous avons proposé enfin une synthèse des avantages de VUML et mis en évidence ses principales limites, dont l'absence du traitement de l'aspect comportemental.

La réalisation de l'état de l'art présenté dans ce chapitre nous a permis d'établir un certain nombre de conclusions.

Les approches à base d'aspects étudiées, malgré leur capacité à proposer des solutions au problème de composition, restent liées au contexte d'application. Ce sont des solutions qui perdent leur généricité au changement du contexte par un passage, par exemple, à la modélisation par points de vue. En effet, elles se basent toutes sur l'identification et la différentiation du modèle métier du modèle transversal (le modèle d'aspect). Ce n'est pas le cas dans le contexte de développement par points de vue où le modèle métier lui-même est décomposé en plusieurs modèles partiels. S'ajoute à cela le fait que les modèles par vues ne font pas de distinction entre les fonctionnalités métier et les fonctionnalités transversales, et donc une identification des points de jonction n'est pas possible. Par conséquent, une adaptation des solutions à base d'aspects au contexte par points de vue n'est pas envisageable.

Dans le chapitre suivant, nous présentons notre proposition de spécification et de composition du comportement suivant une démarche d'analyse conception par points de vue. La contribution principale est une solution basée sur l'observation des modèles partiels du comportement. C'est une solution générique dans son principe. Nous avons mis dans nos perspectives d'expérimenter cette solution au domaine des aspects.

CHAPITRE IV. EXTENSION DE VUML POUR LA MODELISATION DU COMPORTEMENT

Sommaire

IV.1. Introduction

L'objectif de ce chapitre est de répondre à la problématique de spécification comportementale dans le cadre du profil VUML. Nous nous concentrons sur la description du comportement individuel des objets multivue, chacun étant composé d'un ensemble d'objets-vue encapsulant les données spécifiques à chaque acteur. Ce traitement revient à traiter les deux points cruciaux suivants : la spécification des comportements des objets-vue d'une part, et la composition de ces comportements pour former le comportement global de l'objet multivue d'autre part.

La difficulté du problème réside dans le fait que les vues sont développées séparément selon la démarche VUML (cf. section III.4.4), et sont ensuite composées dans la phase de fusion pour former le comportement global des objets multivue. Le compromis qui paraît le meilleur consiste à proposer une approche permettant une liberté la plus grande possible dans le développement des vues, et en même temps à fournir des moyens pour faciliter la fusion.

Pour résoudre ce problème, deux possibilités s'offraient à nous. La première consiste à réutiliser tels quels les mécanismes de spécification du comportement et de la communication entre objets d'UML. La deuxième possibilité consiste à proposer de nouveaux mécanismes, qui étendent ceux d'UML pour s'adapter aux particularités de VUML.

Outre cette introduction, ce chapitre est structuré en deux parties. La section IV.2 développe notre première approche basée sur les concepts d'UML. La section IV.3 présente la deuxième approche basée sur la notion de sonde d'événements, une approche qui représente la contribution principale de cette thèse. Chacune de ces deux parties est terminée par un bilan reprenant les points forts et limites de l'approche traitée. Dans la conclusion du chapitre, nous donnons une comparaison entre les deux approches.

IV.2. Spécification du comportement basée sur les concepts d'UML

Dans cette section, nous décrivons une approche qui consiste à utiliser les mécanismes standard d'UML pour spécifier le comportement des objets-vue VUML et la communication entre les vues. Il s'agit donc de spécifier le comportement des objets-vue par des machines à états communicant à travers des échanges de signaux ou des appels de méthode. Elle propose une technique fondée sur une description séparée puis une coordination des machines à états des objets vues et base.

Pour obtenir une spécification directement exécutable, nous avons utilisé le profil Omega UML pour lequel des outils de simulation et de vérification existent [Ober et al., 06]. Nous proposons également une démarche étendant celle associée à VUML pour expliquer la manière de produire les machines à états proposées. Nous illustrons ces propositions par des extraits de l'étude de cas "Gestion d'une agence de réparation de voitures".

IV.2.1. Principe et définitions

La Figure IV.1 ci-dessous présente la structure d'une classe multivue. La structure statique est représentée par les classes de données stéréotypées par « base » et « view », tandis que le

comportement est représenté par les machines à états (machine-base et machines-vue) associées à ces classes.

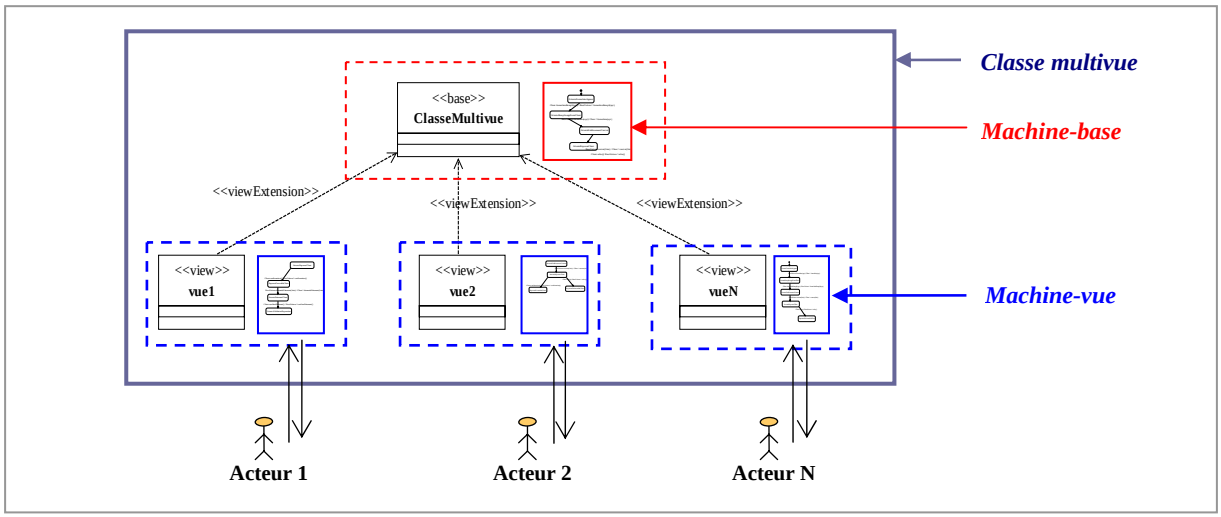


Figure IV.1. Représentation d'une classe multivue

Les machines à états que nous proposons suivent la spécification UML2.0, et sont attachées soit à une base, soit à une vue. La machine-base définit le cycle de vie global d'un objet multivue dans l'objectif d'être partagée par toutes les vues. Elle est constituée d'états qui sont pertinents vis-à-vis de tous les acteurs, et de transitions entre ces états. De ce fait, la machine-base est abstraite et constituée d'états généraux couvrant le cycle de vie de l'objet multivue. Chaque machine-vue partage la structure de la machine-base et la spécialise en rajoutant du comportement qui gère les requêtes provenant de l'acteur associé à la vue. L'ajout de comportement se fait notamment en raffinant les états (abstraits) de la machine-base en des sous-machines (états composés UML). Nous appelons ***machine multivue*** l'ensemble formé d'une machine-base et des machines-vue dépendantes. Le fonctionnement cohérent de la machine multivue s'obtient par la synchronisation des transitions définies dans la machine-base, qui ne peuvent donc être franchies qu'en même temps par toutes les machines-vue.

Nous définissons un **état multivue** comme l'état représentant un objet multivue. C'est un état à plusieurs interprétations selon les acteurs interagissant avec le système, défini par les sous-états suivants :

- Un **état-base** de la machine-base : état abstrait qui représente un point du cycle de vie d'un objet multivue.
- Un ensemble **d'états-vue** constituant la machine-vue : états résultant du raffinement de l'état base en prenant en compte les points de vue des acteurs du système.

A titre d'exemple, dans la Figure IV.2, l'état de la voiture *VoitureEnReparation* (au centre de la figure) est un état multivue, interprété différemment selon chaque type d'acteur. Un mécanicien s'intéresse aux pannes et aux réparations qu'il doit apporter, aux outils nécessaires pour effectuer la réparation et aux pièces de rechange. Tandis qu'un responsable atelier voit la réparation du coté logistique, c'est-à-dire qu'il s'intéresse aux affectations des pistes, à la réservation du matériel, aux affectations des pièces de rechange, etc. Pour un client, les détails techniques d'une réparation sont peu importants, il est plus intéressé par les détails du contrat de réparation, par les frais à engager et

par la date de fin des réparations. Les intérêts du chef de l'agence se focalisent sur l'aspect financier de cette réparation incluant son coût réel, la durée estimée pour l'achèvement des réparations et le contrat à établir avec le client.

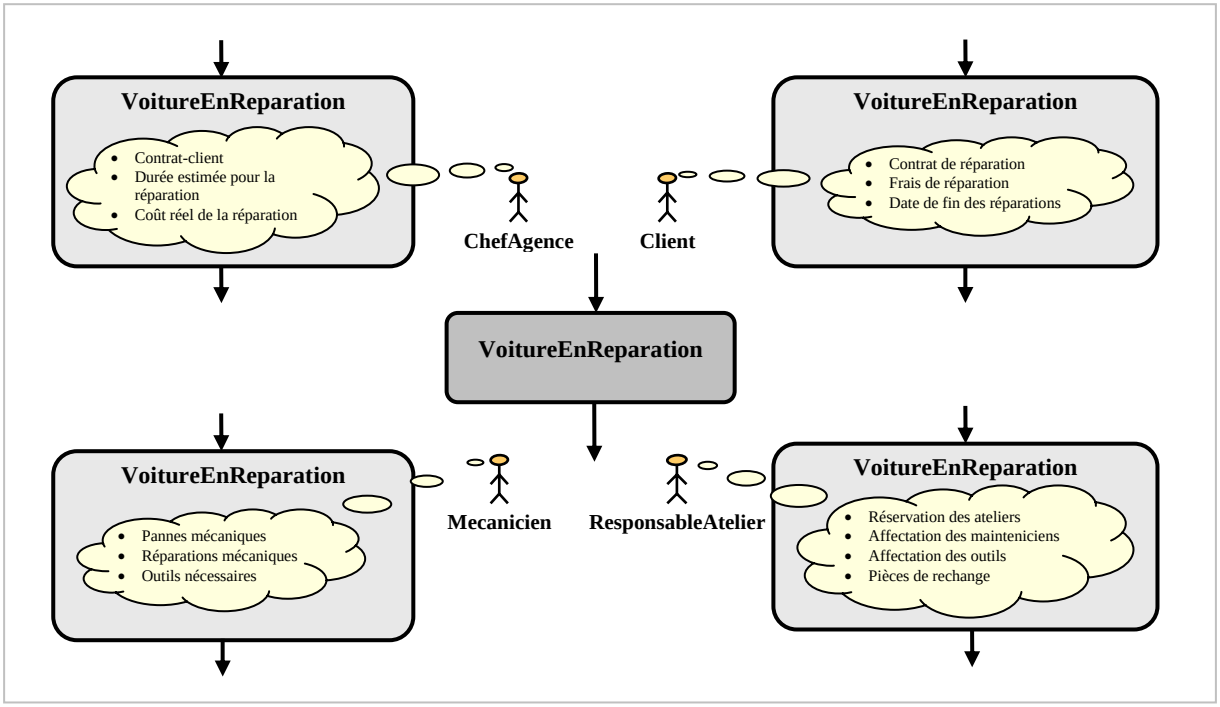


Figure IV.2. Illustration de l'état multivue *VoitureEnReparation*

IV.2.2. Mise en œuvre de l'approche

IV.2.2.1. Principe

La démarche que nous proposons pour construire les machines à états multivue complète la démarche VUML structurelle (cf. section III.4.4) en rajoutant des traitements supplémentaires dans le but d'aboutir aux machines à états : machine-base et machine-vue. Nous décrivons dans cette sous-section les adaptations que nous avons apportées sur la démarche VUML pour prendre en compte les aspects comportementaux de l'approche.

Analyse comportementale globale : le rôle principal de cette phase est l'identification de la machine-base des objets multivue. Elle consiste à extraire les états les plus pertinents de l'objet multivue. Nous obtenons la machine-base sous forme d'un squelette abstrait qui décrit en globalité le comportement attendu de l'objet multivue.

Cette phase d'analyse comportementale est composée des activités suivantes :

1. Identification des classes multivue réactives,
2. Identification des états potentiels pour chaque classe multivue réactive,

3. Construction de la machine à états générale "machine-base" pour chaque classe multivue réactive. Cette machine sera partagée afin qu'elle soit réutilisée lors du développement par point de vue dans la deuxième phase,
4. Ajout du nom de la machine-base dans le glossaire de l'application.

Analyse/conception comportementale par point de vue : Pour chaque acteur, nous faisons une analyse mono-vue des états de la machine-base déjà développée dans la première phase. Nous développons les états qui sont intéressants pour l'acteur en question. A la fin de cette étape nous obtenons les machines-vue de tous les acteurs. Pour chaque classe réactive identifiée dans la phase précédente, le processus de construction d'une machine-vue est le suivant :

1. Etude de la signification de chaque état de la machine base pour l'acteur en question. Selon l'importance de l'état base pour le point de vue traité, l'étude de cet état peut se conclure par le développement d'une sous-machine capturant ainsi les besoins spécifiques de l'acteur,
2. Création de la machine-vue.

Si une classe réactive est identifiée lors du développement par points de vue :

1. Vérification de la présence de la machine-base associée dans le glossaire :
   - Si elle existe : reprise et adaptation en cas de besoin,
   - Sinon : développement et ajout de la machine-base dans le glossaire,
2. Développement de la machine vue.

Fusion/synchronisation comportementale : le rôle de cette phase est de rendre le comportement cohérent en faisant la correspondance entre les machines-vue et la machine-base. Pour chaque classe réactive multivue, le processus suivi est le suivant :

1. Harmonisation de la machine-base (si par exemple un acteur a rajouté un état supplémentaire, qui n'est pas un raffinement des états existants),
2. Etablissement de la communication des machines-vue par synchronisation.

IV.2.2.2. Application sur notre cas d'étude

Comme déjà mentionné, les machines à états proposées pour décrire le comportement des vues suivent le standard UML. Certes, UML fournit une variété de concepts pour spécifier le comportement, possédant tous une notation graphique. Cependant la sémantique donnée à ces éléments reste souvent insuffisante ou carrément indéfinie. En fait, le souci de faire d'UML un standard de modélisation permettant d'analyser/concevoir tout type de domaine a conduit à une sémantique vague et non précise. La syntaxe que nous utilisons pour exprimer les transitions dans les machines à états développées est issue du profil Omega UML [Ober et al., 06]. Ce profil est doté d'une sémantique assez riche offrant un ensemble de mécanismes spécifiques aux aspects communicationnels et d'exécution. Nous utilisons également l'outil IFx associé à ce profil [IFx-site]. Cet outil offre un environnement de simulation et de validation des systèmes Omega UML. Nous nous sommes servis de cet outil pour simuler et valider nos modèles de comportement.

Dans la suite de cette section, nous illustrons l'application de la démarche décrite ci-avant sur notre cas d'étude.

IV.2.2.2.1. Analyse comportementale globale

L'analyse globale structurelle ayant été décrite dans la section III.4.4.2.1, nous décrivons ici les étapes de la phase d'analyse globale comportementale.

Identification des classes multivue réactives : En partant des résultats de l'analyse globale structurelle de l'étude de cas, nous déduisons les classes réactives ainsi que celles potentiellement multivue. En effet, l'analyse poussée des cas d'utilisation ainsi que des besoins de chaque utilisateur – en se servant des diagrammes de séquence et d'activités par exemple – permet d'identifier les classes susceptibles d'être multivue et également celles ayant un comportement réactif. Pour notre cas d'étude, les classes identifiées comme multivue sont : Voiture, Expertise, Panne, Reparation et Contrat. Nous considérons, pour simplifier, que seule la classe Voiture possède un comportement réactif. Les autres classes sont considérées comme des classes de données (statiques). La liste des classes multivue identifiées dans cette étape, ainsi que le classement attribué à chacune d'elles en classes statiques ou en classes réactives, ne sont pas définitifs. D'autres classes multivue pourront apparaître dans la deuxième phase de la démarche lors de l'analyse détaillée par point de vue.

Identification des états potentiels pour chaque classe multivue réactive : Les états potentiels pour un objet sont choisis de manière à couvrir la totalité de son cycle de vie. Pour notre exemple, le cycle de vie d'une instance de la classe Voiture est le suivant : une fois la voiture en panne, son propriétaire établit un contact avec l'agence, et communique les informations nécessaires à l'enregistrement de sa voiture. La chaîne de réparation nominale suivie dans l'agence commence tout d'abord par amener la voiture vers le garage. Ensuite, on procède à l'expertise de la voiture sur les niveaux mécanique et électrique pour détecter les pannes. Suite aux résultats de l'expertise, un contrat de réparation est établi. Les mainteniciens effectuent ensuite la réparation des pannes. La dernière étape est celle du test pour vérifier le bon fonctionnement de la voiture. Ainsi, on aboutit à un cycle de vie nominal englobant les états possibles d'une voiture dans le garage. Nous synthétisons ces états comme suit :

- *EnPanne* : état initial de la voiture ;
- *EnAcheminementGarage* : état qui représente la démarche pour ramener la voiture de l'endroit où elle est tombée en panne vers le garage ;
- *EnProcedureExpertise* : état représentant l'opération d'expertise sur la voiture pour détecter les pannes ;
- *EnProcedureReparation* : après la détection des pannes, la voiture commence la procédure de réparation qui se compose de deux parties essentielles : (i) la négociation du contrat de réparation en se basant sur les résultats de l'expertise, et (ii) la réparation technique représentant l'action de réparation des pannes détectées dans la phase d'expertise ;
- *EnTest* : étape finale avant la sortie du garage pour tester le bon fonctionnement de la voiture ;
- *Sortie* : état représentant la fin du cycle de vie de la voiture dans le garage.

La Figure IV.3 présente la machine à états-base de la classe Voiture. Cette machine résume le cycle de vie d'un objet dans le système et donne l'ordonnancement temporel logique de ses états. Cette machine est ensuite ajoutée dans le glossaire dans l'objectif d'être partagée et réutilisée dans la deuxième phase de la démarche par les concepteurs par point de vue.

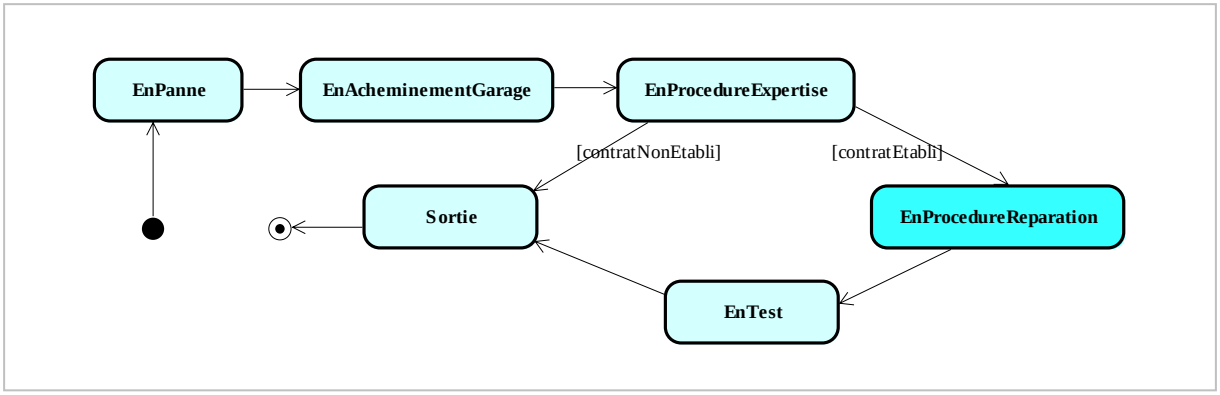


Figure IV.3. Machine-base associée à la classe Voiture

IV.2.2.2.2. Analyse/Conception comportementale par point de vue

Après l'établissement du modèle structurel selon un point de vue particulier (cf. section III.4.4.2.2), on démarre la phase de spécification comportementale par point de vue. Elle consiste à faire une analyse mono-vue des états de la machine-base identifiée dans la première phase. Chaque état-base peut donner naissance à une sous-machine le raffinant et le spécialisant aux besoins du point de vue traité. Nous proposons de restreindre l'illustration de notre exemple :

- à une partie de la machine-base. L'état que nous allons détailler est l'état *EnProcedureReparation* durant lequel la voiture subit une série d'actions afin qu'elle retrouve son état normal de fonctionnement,
- aux deux acteurs client et chef de l'agence.

Point de vue *Client* : En nous basant sur les diagrammes de séquence développés pour le point de vue *Client* (cf. section III.4.4.2.2), notamment ceux qui traitent la procédure de réparation et de négociation du contrat, nous aboutissons à la partie de la machine à états *SM_Voiture_Client* concernant l'état *EnProcedureReparation* (Figure IV.4).

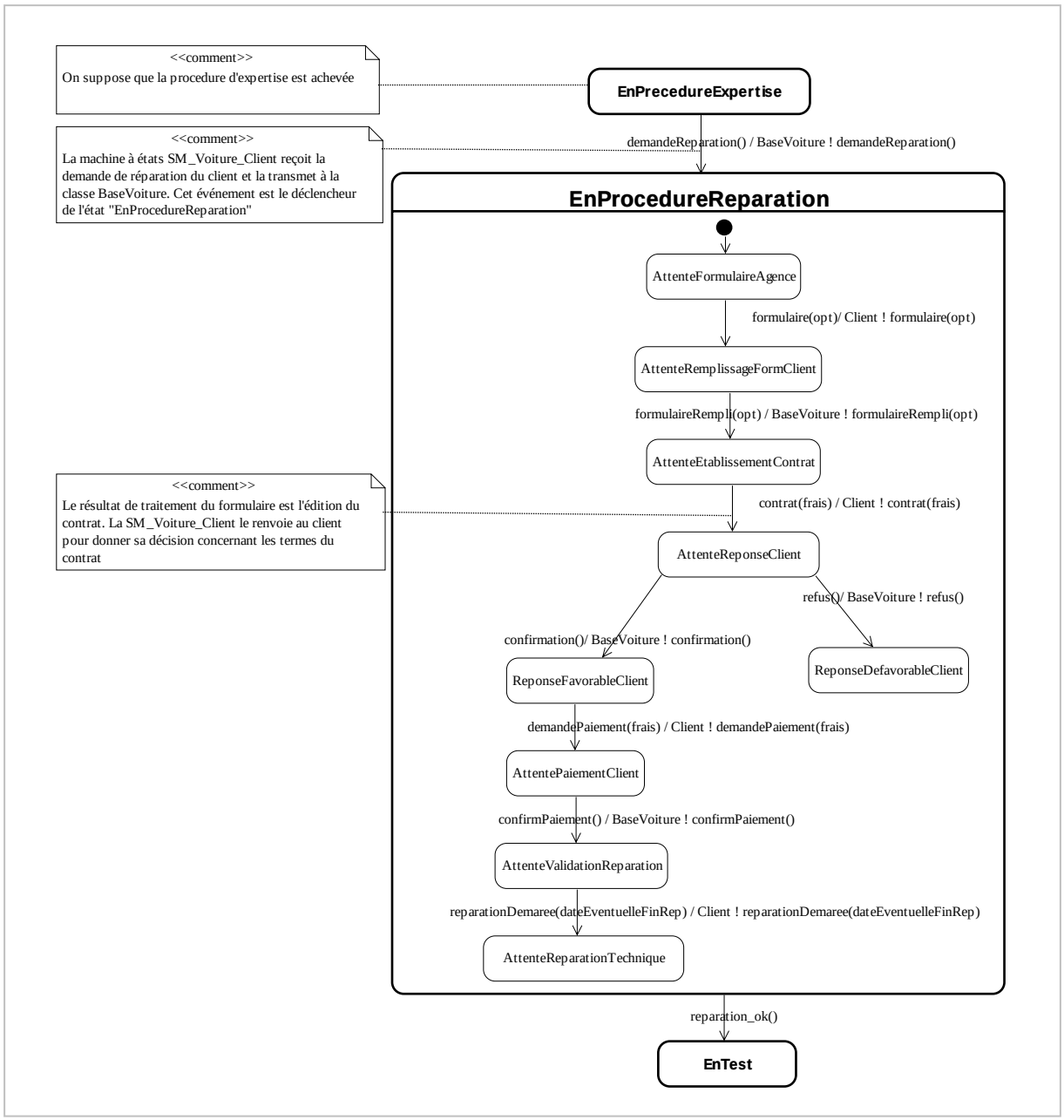


Figure IV.4. Raffinement de l'état EnProcedureReparation pour le point de vue Client

La transition dans la machine-vue *SM_Voiture_Client* de l'état *EnProcedureExpertise* vers l'état *EnProcedureReparation* est déclenchée par le signal **`demandeReparation()`** provenant de l'acteur client. Or dans cette étape de la démarche, l'analyse/conception par point de vue est réalisée d'une manière décentralisée. De ce fait, la machine à états *SM_Voiture_Client* retransmet ce signal vers la machine-base sans se soucier vers quel ou quels objets ce signal sera transmis. Le même principe est appliqué dans le cas où un acteur attend un signal provenant d'une autre entité. Si c'est le cas, le développeur de ce point de vue suppose que le signal provient de la machine-base. C'est l'exemple du signal **`formulaire(opt)`** qui déclenche la transition entre l'état *AttenteFormulaireAgence* et l'état *AttenteRemplissageFormClient* sur la Figure IV.4.

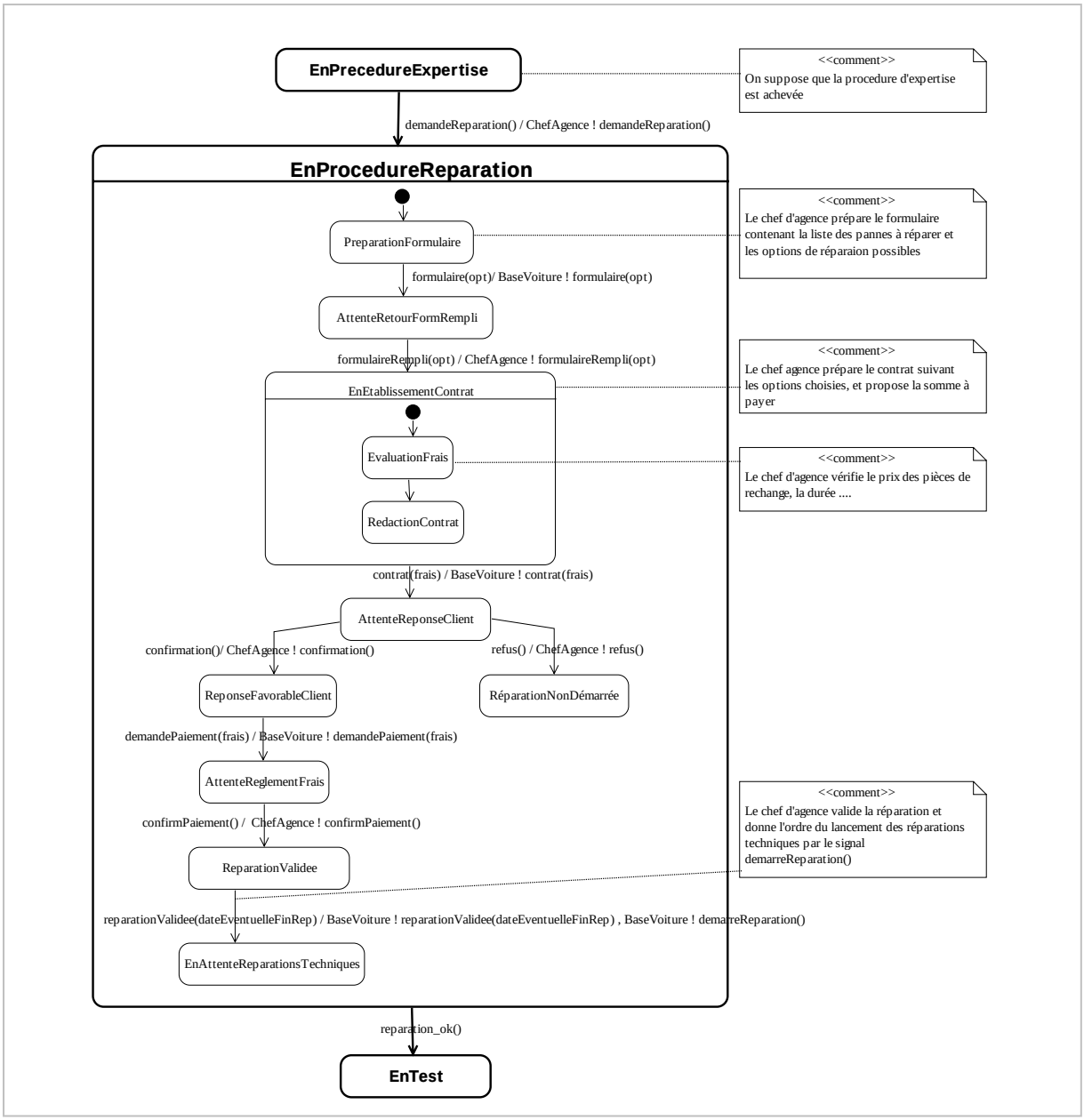


Figure IV.5. Raffinement de l'état EnProcedureReparation pour le point de vue ChefAgence

Point de vue du *ChefAgence* : Le raffinement de l'état *EnProcedureReparation* est représenté dans la Figure IV.5. Le même principe sur la centralisation des communications dans la machine-base est appliqué. La *SM_Voiture_ChefAgence,* quand elle reçoit le signal `demandeReparation()` provenant de la machine-base, le renvoie vers l'acteur chef d'agence. Ensuite, le chef d'agence prépare un formulaire contenant la liste des pannes à réparer, ainsi que les choix qui accompagnent la réparation de chaque panne. Une fois le formulaire prêt, le chef d'agence l'envoie à la *SM_Voiture_ChefAgence* (c'est le signal `formulaire(opt)` qui déclenche la transition vers l'état *AttenteRetourFormRempli*).

IV.2.2.2.3. Fusion comportementale

L'objectif de cette étape est d'assurer la cohérence dans le comportement des objets multivue en synchronisant les différentes machines à états associées à une classe multivue. Le principe de la synchronisation consiste à assurer la communication entre les différentes machines-vue à travers la machine-base. En fait, la synchronisation revient à alimenter la machine-base (voir la Figure IV.3) par des messages dans l'objectif de créer la liaison entre les machines-vue, c'est-à-dire à faire la correspondance entre le demandeur de l'information et son fournisseur.

Par exemple, dans la machine-vue de la voiture associée au point de vue *Client* (voir Figure IV.4), le signal *demandeReparation()* a été envoyé à la machine-base (car le développeur du point de vue *Client* ignorait qui va traiter ce signal). La machine-vue du chef d'agence (voir Figure IV.5) attend ce signal pour franchir la transition de l'état *EnProcedureExpertise* vers l'état *EnProcedureReparation*. Le premier message de synchronisation montré sur la Figure IV.6 suivante vise la réalisation de cette correspondance. Le travail de synchronisation est poursuivi en suivant la même démarche jusqu'à produire la machine-base finale.

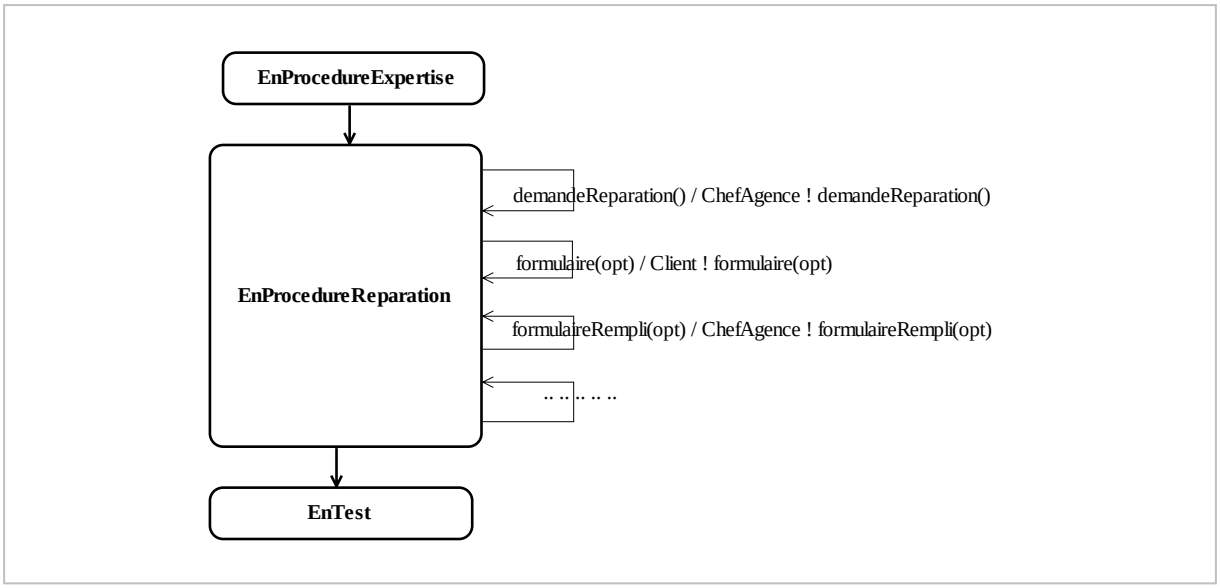


Figure IV.6. Ajout de messages de synchronisation dans la machine-base pour l'état EnProcedureReparation (Extrait)

IV.2.3. Bilan de l'approche basée sur les concepts UML

Comme nous venons de le voir, cette première approche est basée uniquement sur les concepts d'UML. Le résultat essentiel tiré de cette approche est l'identification, pour une classe multivue donnée, de deux types de machines à états particulières attachées soit à une classe « base », soit à une classe « vue ». Ces machines à états, qui suivent la spécification UML2.0 [OMG-UML], ont pour rôle de capturer le comportement dynamique des instances de la classe multivue considérée [Lakhrissi et al., 07], [Lakhrissi et al., 08a]. Une machine-vue représente le cycle de vie d'un objet-vue, tandis qu'une machine-base a pour but de créer la cohérence et la coordination entre les machines-

vue, et de spécifier le comportement commun aux acteurs. On appelle ***machine multivue*** l'ensemble formé d'une machine-base et des machines-vue dépendantes.

Cette approche répond à la problématique de la manière suivante : pour favoriser un développement indépendant dans la deuxième phase de la démarche, il était indispensable de que cette phase soit guidée par un modèle établi a priori ; d'où la proposition de la notion de machine-base. La machine-base est développée lors de la phase de modélisation globale pour spécifier le comportement commun aux acteurs. Cette dernière est considérée comme un patron à respecter lors du développement des comportements par points de vue. Dans la phase de modélisation décentralisée on procède à la description séparée des machines-vue associées aux objets-vue suivant les états composant la machine-base. Dans la phase de fusion, la composition des comportements partiels des différents points de vue est réalisée en ajoutant les échanges de signaux qui permettent la coordination des machines à états des objets-vue et de l'objet-base.

Toutefois, l'expérience avec cette approche, basée uniquement sur les concepts d'UML, lors d'un passage à l'échelle montre des limites :

- ***par rapport à la spécification du comportement.*** Nous avons rencontré des difficultés pour assurer l'indépendance dans le développement des vues car ces dernières peuvent être fortement intercouplées. Lors d'un passage à l'échelle, cette situation met la démarche en péril et rend difficile sa mise en œuvre sans altérer le développement des autres vues afin de recueillir les informations manquantes. S'ajoute à cela le problème de devoir identifier, assez tôt, les classes multivue et leurs machines-base, lors de la phase d'analyse globale. En fait, proposer une machine à état couvrant le cycle de vie d'un objet demande une étude approfondie des cas d'utilisation de l'objet multivue. Or pour un système de grande taille, produire une telle machine n'est pas envisageable.
- ***par rapport à la composition du comportement.*** L'intégration des machines-vue développées séparément peut requérir de nombreuses modifications et adaptations au niveau des machines-vue et de la machine-base. Lors d'un passage à l'échelle, un travail conséquent doit accompagner l'opération de composition pour assurer la cohérence de l'ensemble.

Ces deux limites posent un problème d'applicabilité de l'approche. Nous avons en conséquence opté pour une autre solution proposant de nouveaux mécanismes spécifiques à la modélisation et à la composition des comportements des objets multivue. Pour cela, nous avons défini la notion d'observation d'événements qui permet de spécifier des communications implicites entre les objets-vue. Ceci permet de découpler des spécifications qui sont a priori fortement interconnectées, de les concevoir indépendamment selon les préconisations de la méthode VUML, puis de les intégrer sans avoir à les modifier.

IV.3. Spécification du comportement basée sur les sondes d'événements

Nous présentons dans cette section la deuxième approche basée sur une communication implicite entre les vues au travers des observations d'événements. Nous signalons qu'au niveau de la terminologie, la notion de machine-base n'existe plus dans cette approche, par contre le terme de machine-vue garde sa signification et représente la machine à états d'une classe du système selon la vue considérée.

Nous proposons une technique de spécification de comportement, que nous appelons ***sonde d'événements***, basée sur l'observation des événements du système en exécution. Elle répond aux problèmes liés à la spécification et à la composition des comportements des différents modèles-vue. En effet, les comportements des modèles-vue évoluent d'une manière indépendante les uns des autres et chacun d'eux utilise et génère un ensemble d'événements. Notre proposition consiste, dans la phase d'intégration des vues, à utiliser l'observation des événements (et des informations relatives à ces événements) comme moyen de communication entre les vues. Sur la base de cette proposition, nous avons réalisé une extension du profil VUML, appelée VxUML, permettant la spécification comportementale des objets multivue [Ober et al., 08a ; Ober et al., 08b]. Il s'agit d'intégrer la notion d'observation dans le profil VUML en identifiant les types d'événements adéquats et en fixant une représentation compatible avec UML des objets de type sonde correspondants.

Cette section est composée de quatre parties. La première définit le concept d'observation basé sur la notion de sonde d'événements. La deuxième partie définit les concepts de base que nous avons développés pour manipuler les sondes. La troisième partie explique comment la notion de sonde et les concepts associés sont utilisés dans UML. Nous décrivons dans cette section également le nouveau profil VxUML, extension du profil VUML intégrant la technique de description du comportement utilisant les sondes d'événements. La quatrième partie explique la manière d'expérimenter les éléments du nouveau profil pour définir et composer le comportement des objets multivue. La section se termine par le bilan et les conclusions tirées de cette solution.

IV.3.1. Principes

IV.3.1.1. Notion de sonde d'événements

Une exécution d'un système est considérée comme une trace d'événements. Un événement désigne le plus petit changement dans l'état de ce système. Il représente l'exécution d'une action atomique indiquant l'occurrence d'un fait particulier. Les événements sont de plusieurs types : appel d'une méthode d'un objet (communication synchrone), réception d'un signal (communication asynchrone), changement d'état d'une condition booléenne, déclenchement d'une transition dans une machine à états, entrée dans un état, etc.

Nous définissons le terme de *sonde d'événements* comme étant un élément de modélisation qui permet d'identifier un événement ou une séquence d'événements et de les utiliser dans la spécification du comportement lors de la phase de conception. Cette utilisation peut prendre plusieurs formes telles que la recherche et la détection d'un comportement donné dans le système, le contrôle de certains états critiques, le déclenchement de comportements suite à la réalisation d'autres comportements, etc. Comme exemples de sondes, on peut citer : (i) sonde permettant de faire référence à tous les événements de type « création d'un objet d'une classe donnée "C" », (ii) sonde permettant de détecter et de faire référence à tout événement de type « envoi d'un signal de type "S" dans le système », (iii) sonde détectant l'événement « réception du signal de type "S1" par une instance de la classe "C" », etc. D'une manière générale, une sonde se définit par rapport à un type d'événement particulier (création d'objet, envoi de signal, appel d'opération, etc.).

Chaque événement à l'exécution se produit dans un *contexte* particulier. Les sondes permettent d'accéder aux informations relatives à ce contexte, qui dépendent du type de l'événement. Par

exemple, pour un événement de type « création d'objet », les informations de contexte sont : la classe de l'objet créé, l'identifiant de l'objet créé, l'identifiant de l'objet « parent » (qui l'a créé), etc. Ces données sont stockées comme attributs de la sonde.

Au niveau de la modélisation, définir une sonde revient à définir une instance d'un type de sonde prédéfini (une classe de librairie). En plus du type, la définition peut préciser une condition (ou plusieurs) qu'un événement doit satisfaire pour activer la sonde. Cette condition porte sur les informations contextuelles de l'événement (les attributs de la sonde), elle est mise en œuvre en se basant sur la technique de projection de sondes présentée dans la section IV.3.2.4. La sémantique des sondes précise que, quand l'occurrence d'un événement active une sonde, les attributs de celle-ci sont mis à jour avec les données contextuelles de l'événement.

En résumé, l'objectif des sondes est de permettre l'accès aux événements au cours d'une exécution, c'est-à-dire d'accéder aux données correspondant à ces événements (l'identifiant de l'élément déclenchant l'événement, l'identifiant de l'élément cible de l'événement, les paramètres transmis, etc.) ainsi qu'à leurs méta-données (telles que la classe des objets concernés par l'événement, …). Une fois la sonde d'un événement donné déclenchée dans le système en exécution, les attributs de la sonde stockent les données et les méta-données liées à cet événement.

IV.3.1.2. Fonctionnement des sondes

L'intérêt de la définition d'une sonde est de pouvoir identifier des événements et de les utiliser dans la spécification du comportement des objets. Ceci est réalisé par une construction permettant à un objet d'attendre l'apparition d'un événement observé par une sonde « obs ». Cette construction, que nous notons *wait(*obs*),* est définie comme un nouveau type de déclencheur de comportement (Trigger) utilisable sur les transitions des machines à états. La Figure IV.7 suivante explique le principe de communication inter-objets basé sur les sondes d'événements.

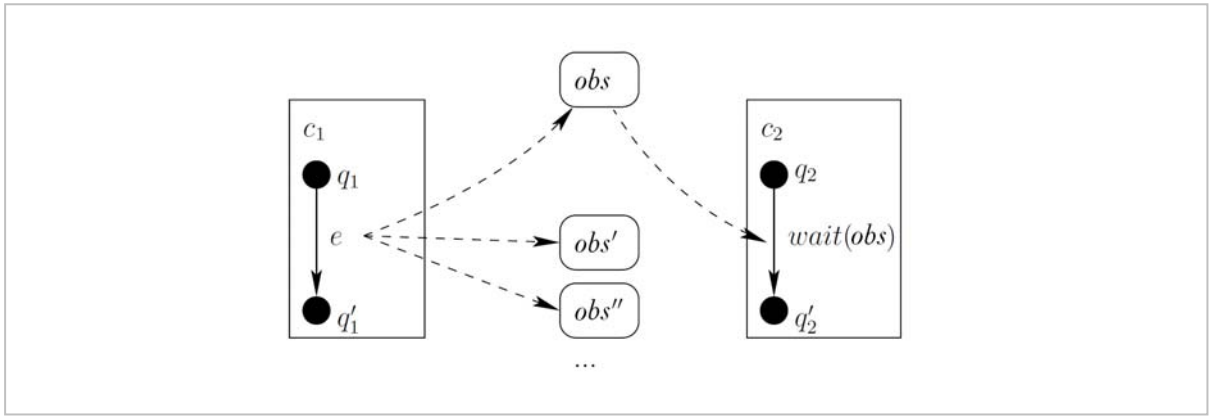


Figure IV.7. Principe de la communication basée sur le concept de sonde

Quand un objet c_1 exécute une action, cela produit un événement « e » qui peut être observé par une ou plusieurs sondes (« obs », « obs' », « obs" » dans la Figure IV.7). Un autre objet c_2 peut observer l'événement en utilisant une transition déclenchée par *wait(obs)*. La communication entre c_1 et c_2 est implicite dans la mesure où dans le modèle développé aucun mécanisme de communication

explicite n'est spécifié par l'utilisateur pour informer c_2 de la réalisation de l'événement « e » de l'objet c_1. La transition gardée par « wait(obs) » ne peut être franchie que si c_2 est déjà dans l'état q_2 quand « e » s'est produit. c_2 devient donc exécutable immédiatement.

En faisant une analogie avec les exceptions dans les langages à objets, on peut faire le parallèle entre les objets de type sonde décrits ici et les objets utilisés pour transmettre des exceptions prédéfinies levées par exemple par une machine virtuelle Java (par exemple : ArithmeticException, NullPointerException…). L'activation d'une sonde et la mise à jour de ses attributs se font implicitement, comme pour la levée d'exceptions prédéfinies. Par contre, la sonde peut ensuite être utilisée ou pas pour déclencher un comportement.

IV.3.2. Définition des concepts de base pour la manipulation des sondes

Les objectifs visés avec la notion de sonde d'événements ne peuvent pas se concrétiser en utilisant les éléments de modélisation existants. Dans cette section, nous définissons les concepts nécessaires pour pouvoir spécifier et déterminer ***la portée d'une sonde***. En fait, le champs couvert par une sonde doit être concis et bien déterminé pour pouvoir identifier des comportements fins et précis. Pour réaliser cela, nous avons d'abord identifié un ensemble de familles de sondes assorties avec les catégories d'événements usuels manifestés dans une exécution. Puis, par la notion de projection, on peut raffiner ces types pour cibler un comportement précis. Ensuite, nous avons défini le principe de dérivation de sondes. C'est un mécanisme qui permet de créer de nouvelles classes de sondes avec des attributs supplémentaires, mais qui observent toujours le même type d'événement que la classe mère. Nous offrons un autre mécanisme dans le but de personnaliser une classe de sondes qui est la composition. En effet le concepteur peut créer de nouvelles classes composites, classes capables d'observer plusieurs types d'événements à la fois.

Dans cette section nous définissons les concepts de base nécessaires à la manipulation des sondes à savoir : (1) quels sont les types de sondes élémentaires (cf. sous-section IV.3.2.1) ; (2) comment est définie la structure statique d'une sonde (cf. sous-section IV.3.2.2) ; (3) comment déclarer et instancier les sondes élémentaires (cf. sous-section IV.3.2.3) ; (4) comment les dériver pour obtenir des sondes spécialisées (cf. sous-section IV.3.2.4) ; (5) comment composer les sondes (cf. sous-section IV.3.2.5) ; et finalement (6) comment sont-elles utilisées par les classes du système (cf. section IV.3.2.6).

IV.3.2.1. Types de sondes élémentaires : la bibliothèque *ProbeLibrary*

En étudiant les types d'événements pouvant être produits lors de l'exécution d'un système, nous avons identifié trois familles de sondes élémentaires (cf. Figure IV.8) : (i) les sondes d'événements de communication, tels que l'émission/réception de signaux et l'appel/retour d'opérations ; (ii) les sondes d'événements liés aux changements dans la structure du système, tels que la création/destruction d'objets et la création/destruction de liens entre objets ; et (iii) les sondes de modification de données.

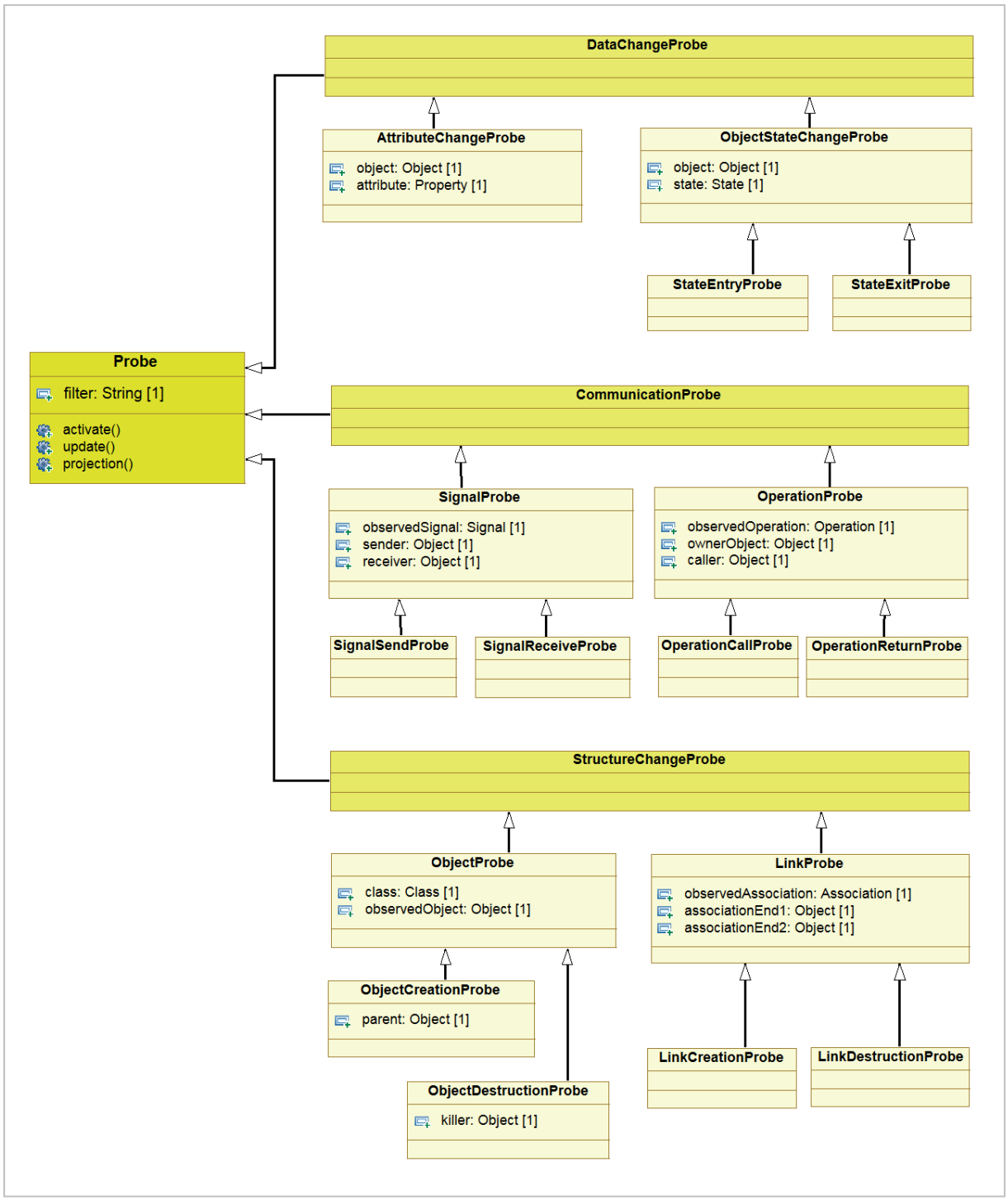


Figure IV.8. Types de sondes : la bibliothèque ProbeLibrary

Ces types de sonde élémentaire se situent au niveau de modélisation M1 selon la terminologie d'MDA [Soley et al., 00]. Nous les avons définis dans une librairie ***ProbeLibrary*** offrant ainsi aux concepteurs la possibilité de les utiliser dans les modèles de conception comme des classes prédéfinies. Chaque type de cette bibliothèque peut être instancié pour donner naissance à des sondes concrètes. Ces types, comme mentionné ci-avant, peuvent faire l'objet d'une personnalisation du contexte par le moyen de projection, de dérivation et de composition.

Par leur nature particulière, les sondes peuvent manipuler des données de niveau modèle (niveau M1) ou de niveau méta-modèle (niveau M2). Si on prend par exemple une sonde de type *ObjectProbe*, elle manipule le type *observedObject* qui est un objet de niveau M1, et le type *class* qui désigne sa classe d'appartenance qui est un élément du niveau M2. En conséquence, pour manipuler les sondes, il faut utiliser un langage qui supporte la réflexivité, c'est-à-dire un langage capable de manipuler les données des niveaux M1 et M2. UML ne fait pas de préconisation à ce sujet, mais il est en principe possible d'utiliser des éléments de méta-modèle dans un modèle UML. Pour l'implantation, on peut citer Java comme langage de niveau M1 qui présente un support, bien que limité, pour le niveau M2.

Nous donnons dans les trois sections qui suivent des définitions pour les trois types de sondes élémentaires proposés. Notons qu'à ce stade, il est difficile de donner des exemples concrets sur l'utilisation des différentes sondes. Nous donnerons des exemples détaillés de ces types de sondes après la présentation de l'opération de projection dans la section IV.3.2.4.

IV.3.2.1.1. Sondes de changement structurel du système

Les sondes de changement de structure (*StructureChangeProbe*) permettent l'observation des événements liés aux modifications structurelles dans le système. Ainsi, ces sondes peuvent faire référence à la vie des objets (création/destruction d'un objet), mais aussi aux relations structurelles entre entités, telles que la création et la destruction de liens.

► Sondes de vie d'objets (ObjectLifeProbe)

- *ObjectCreationProbe* : ce type de sonde permet de détecter et de référencer les créations de nouveaux objets dans le système en exécution.
- *ObjectDestructionProbe* : ce type de sonde permet de détecter les destructions d'objets dans le système en exécution.

  Une fois la sonde activée, elle stocke les variables du contexte de l'objet concerné (dans le premier cas l'objet créé, et dans le deuxième cas l'objet détruit) pour qu'elles soient utilisées par l'objet utilisant cette sonde.

  Exemple : pour contrôler le nombre total des objets du système, on peut déclarer deux sondes de type *ObjectLifeProbe* : la première (obs1) pour référencer toutes les créations d'objets, la deuxième (obs2) pour référencer les destructions d'objets dans le système. La figure ci-dessous illustre la déclaration des deux sondes (Figure IV.9-a), ainsi que leur utilisation par l'objet contrôleur (Figure IV.9-b). La variable nbObjets est utilisée pour stocker le nombre d'objets dans le système.

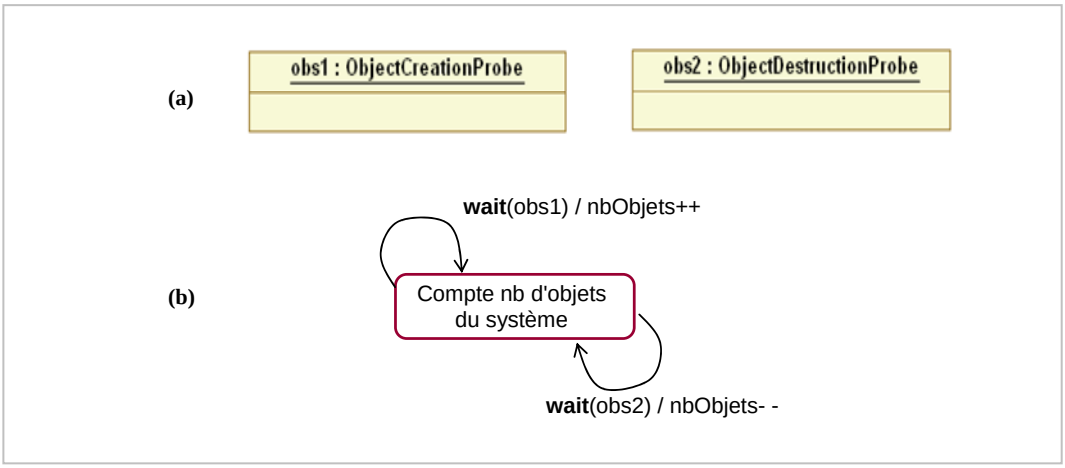


Figure IV.9. Exemples de sondes ObjectLifeProbe

► **Sondes de liaisons (LinkProbe)**

- *LinkCreationProbe* : ce type de sonde permet de détecter les créations de nouvelles liaisons entre objets du système.
- *LinkDestructionProbe* : ce type de sonde permet de référencer les destructions de liaisons entre objets du système.

IV.3.2.1.2. Sondes de communication

Les sondes de communication (*CommunicationProbe*) permettent l'observation des événements liés aux interactions explicites entre objets. Ces sondes peuvent observer et référencer les échanges de signaux entre les entités du système. Elles peuvent également observer et référencer les appels de méthodes et leurs retours effectués entre les différents objets du système.

► **Sondes de signaux (SignalProbe)**

- *SignalSendProbe* : ce type de sonde permet de référencer les émissions de signaux entre objets du système en exécution.
- *SignalReceiveProbe* : ce type de sonde permet de référencer les réceptions de signaux par un objet du système en exécution.

► **Sondes d'opérations (OperationProbe)**

- *OperationCallProbe* : ce type de sonde permet de référencer les appels d'opération entre objets du système en exécution.
- *OperationReturnProbe* : ce type de sonde permet de référencer les retours d'opération.

IV.3.2.1.3. Sondes de modification de données

Les sondes de modification de données (*DataChangeProbe*) concernent deux types d'événements : les événements liés aux changements de la valeur d'un attribut du système, et les événements liés aux changements d'états dans une machine à états d'un objet.

► Sondes de changement des valeurs d'attribut (AttributeChangeProbe)

Ce type de sonde permet l'observation des changements de valeurs d'attributs des objets du système en exécution. La sonde devient active à chaque changement de valeur d'un attribut de l'un des objets du système. Ce type de sonde s'avère dangereux à utiliser directement dans le système. Si la sonde s'active pour tout changement d'attribut, cela risque d'amener la sonde dans une boucle infinie, car l'activation de la sonde elle-même se fait par changement d'attribut. La notion de projection, que nous présentons dans la section suivante, permet de focaliser la sonde sur un attribut ou un ensemble d'attributs bien identifiés.

► Sondes de changement d'états d'objets (ObjectStateChangeProbe)

Ce type de sonde permet de référencer les changements d'état des objets du système en exécution. On différencie les sondes de détection d'entrée dans un nouvel état, de celles de la détection de sortie d'état.

- *StateEntryProbe* : ce type de sonde permet de référencer les entrées dans de nouveaux états des objets du système en exécution.
- *StateExitProbe* : ce type de sonde permet de référencer les sorties d'états des objets du système en exécution.

IV.3.2.2. Structure des classes prédéfinies de sondes

Un type de sondes est un élément de modélisation particulier capable de stocker et manipuler des données et des méta-données. Nous définissons cet élément comme un *classifier* susceptible d'avoir des attributs, des opérations et des machines à états (pour décrire le comportement de la sonde). Nous faisons la différence entre deux catégories de sondes : les sondes élémentaires et les sondes composées à travers l'attribut *isComposite* (nous représentons cet attribut comme une *tagged value* dans la section IV.3.3.2 sur le profil Probe_profile). Une sonde prédéfinie possède un ensemble d'attributs et d'opérations prédéfinis que nous présentons ci-dessous :

Attributs prédéfinis

Les attributs d'une sonde permettent de stocker des données et des méta-données en relation avec l'événement observé par la sonde. Les attributs prédéfinis communs à l'ensemble des types de sondes sont les suivants :

- attributs spécifiques au type de la sonde : chaque type de sonde présent dans la bibliothèque *ProbeLibrary* (cf. section IV.3.2.10) a une liste d'attributs de type méta-donnée. Leur rôle principal est de stocker les informations en liaison avec le contexte de

l'événement observé par la sonde, telles que la classe mère d'un objet, l'association reliant deux classes, etc.

- filter : attribut qui décrit les conditions booléennes sur les méta-données de l'événement observé par la sonde. Le langage utilisé pour exprimer ces contraintes est le langage OCL.

Attributs personnalisés

Le concepteur peut déclarer de nouveaux attributs pour stocker des informations supplémentaires sur l'état du système au moment de l'apparition de l'événement observé. Cela est réalisé par le mécanisme de dérivation de sonde (cf. sous-section IV.3.2.5).

Opérations prédéfinies

Les deux catégories (élémentaire, composite) de sonde ont les opérations prédéfinies suivantes :

- **projection() :** c'est une opération qui permet de définir les conditions d'activation d'une sonde. Selon la catégorie de la sonde, l'opération *projection()* change de comportement :
  - dans le cas d'une sonde élémentaire, l'opération *projection()* prend comme condition d'activation la contrainte booléenne spécifiée par l'attribut *filter* ;
  - dans le cas d'une sonde composée, on associe à l'opération *projection()* une machine à états pour exprimer les règles d'assemblage des sondes qui font partie de la sonde composée.
- **activate() :** c'est une opération qui s'exécute automatiquement une fois la sonde activée, c'est-à-dire quand :
  - le filtre sélectionne l'événement observé dans le cas des sondes élémentaires ;
  - la machine à états associée à l'opération *projection()* arrive à l'état final dans le cas des sondes composées.

  Le principal objectif de l'opération *activate()* est d'appeler la méthode *update()* pour mettre à jour les attributs de la sonde, et également débloquer la transition bloquée par le trigger *wait*.
- **update() :** comme mentionné ci-dessus, la méthode *update()* est lancée une fois les contraintes de la projection satisfaites. C'est une méthode destinée à la mise à jour des attributs de la sonde. Si une sonde est dérivée, il faut redéfinir la méthode *update()*.

Opérations personnalisées

Le concepteur, par le moyen de dérivation, peut personnaliser les opérations de la classe de sondes utilisée, pour :

- S'adapter aux attributs personnels ajoutés par dérivation. Dans ce cas le concepteur doit mettre à jour l'opération *update()* pour qu'elle puisse prendre en considération la mise à jour des attributs ajoutés, comme montré dans la section traitant la dérivation de sondes.

- Ajouter de nouvelles opérations qui seront exécutées au moment de l'activation de la sonde. Dans ce cas le concepteur doit ajouter leurs déclarations au sein de la sonde dérivée, puis les appeler dans l'opération *activate()*.

IV.3.2.3. Déclaration et instanciation d'une sonde

La déclaration d'une sonde se fait d'une manière indépendante par rapport aux entités du système. C'est-à-dire qu'une classe de sondes ne peut pas être modélisée comme les classes habituelles et participer dans des associations avec les entités du système.

Une fois la structure de la classe de sondes définie (une sonde prédéfinie ou une nouvelle classe créée par dérivation ou par composition), le concepteur peut instancier cette classe pour l'utiliser dans ses modèles de conception.

Nous illustrons sur la Figure IV.10 un exemple d'instanciation de la classe de sonde *SignalSendProbe*. La sonde *reparationOkObs* de type *SignalSendProbe* est instanciée dans le modèle pour détecter les envois de signaux. Si aucune contrainte n'est listée dans l'attribut *filter*, la sonde deviendra active lors de tout envoi de signal dans le système ; sinon la sonde sélectionnera les événements qui répondent à la contrainte *filter* (voir section suivante sur la projection). L'activation signifie que les attributs de la sonde sont mis à jour avec les paramètres de l'événement et que les objets en attente dans un *wait* sur la sonde sont débloqués. Cela se fait implicitement par l'exécution de la méthode *activate()*.

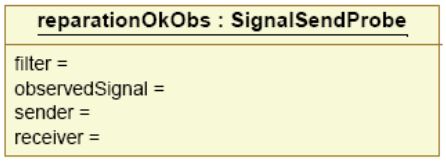


Figure IV.10. Exemple d'instanciation de la classe de sondes SignalSendProbe

IV.3.2.4. Projection des sondes

Chaque type de sonde est associé à un type d'événement et par conséquent la sonde s'active pour toute apparition de ce type d'événement dans le système. Cette propriété n'est pas toujours souhaitée, car parfois nous avons besoin de filtrer les événements sur la base des informations du contexte. Ceci revient à appliquer des contraintes supplémentaires qui doivent être satisfaites au moment de l'activation des sondes. Si on prend par exemple les deux types de sondes *SignalSendProbe*/*SignalReceiveProbe*, ils permettent de référencer les émissions/réceptions de signaux entre objets du système en exécution. Si aucune condition n'est spécifiée dans la définition de la sonde, la sonde sera déclenchée pour toute émission/réception de signal dans le système. En appliquant l'opération de projection, il est possible de restreindre le champ d'activation de la sonde à un contexte particulier.

IV.3.2.4.1. Principe d'une projection

Nous avons muni les sondes d'une opération notée *projection()* (cf. Figure IV.8). Cette opération permet d'appliquer des contraintes supplémentaires sur les conditions d'activation d'une sonde. Elle définit le contexte de l'observation en ajoutant des conditions à satisfaire par les événements visés par le type de la sonde. L'opération de projection prend les contraintes depuis l'attribut *filter* de la sonde. Cet attribut permet d'exprimer dans une chaîne de caractères des conditions booléennes sur les données et les méta-données des événements observés. Nous utilisons le langage OCL pour exprimer ces contraintes.

La Figure IV.11 donne un aperçu de la projection de la sonde *SignalSendProbe*. Le filtre de la sonde, *filter*, est une conjonction de trois contraintes OCL à vérifier par les événements de type envoi de signal. La première contrainte vérifie que le type du signal observé doit être de type *voitureReparee* ; la deuxième spécifie que l'objet émetteur est de type Mecanicien ; et la troisième contrainte vérifie que l'objet récepteur est de type ResponsableAtelier.

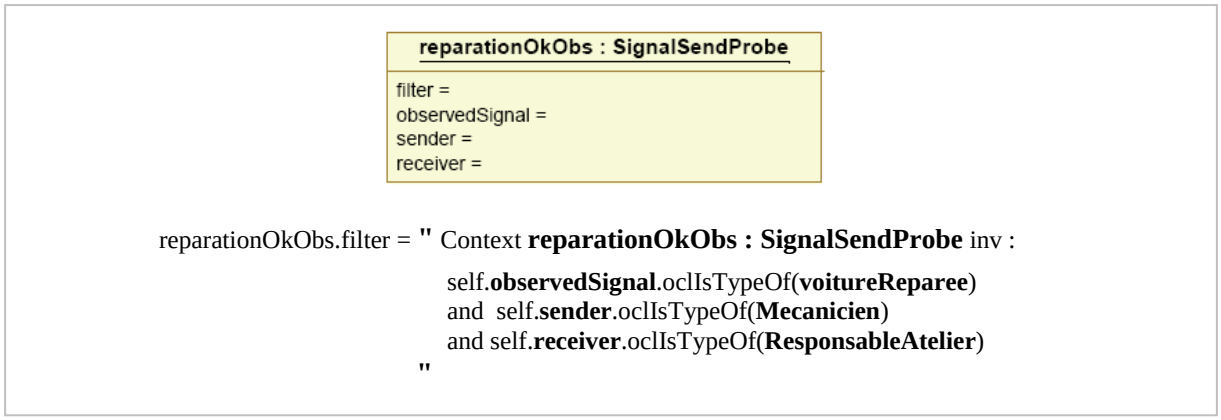


Figure IV.11. Exemple de filtre : projection de la sonde SignalSendProbe

La sonde est *activée* par un événement uniquement si les paramètres de l'événement satisfont la condition *filter*. L'activation signifie que les attributs de la sonde sont mis à jour avec les paramètres de l'événement et que les objets en attente dans un *wait* sur la sonde sont débloqués. Cela se fait implicitement par l'exécution de la méthode *activate()*.

IV.3.2.4.2. Exemples d'application de la projection

Dans cette section, nous présentons des exemples concrets d'application de la projection sur les types de sondes élémentaires. La Figure IV.12 présente le modèle de conception à partir duquel sont tirés des exemples pour illustrer la projection de sondes. Il s'agit d'un diagramme de classes extrait de notre cas d'étude "Gestion d'une agence de réparation de voitures". Pour ne pas rendre les exemples d'illustration des sondes complexes, ce digramme de classes n'est pas développé suivant l'approche par points de vue. C'est un diagramme simple qui représente une partie de la structure de l'application, mais qui est suffisant pour alimenter les exemples présentés dans la suite de ce chapitre. Ce diagramme est accompagné de deux packages : le premier contient la déclaration de signaux et le deuxième contient les déclarations de sondes qui seront utilisées dans les exemples.

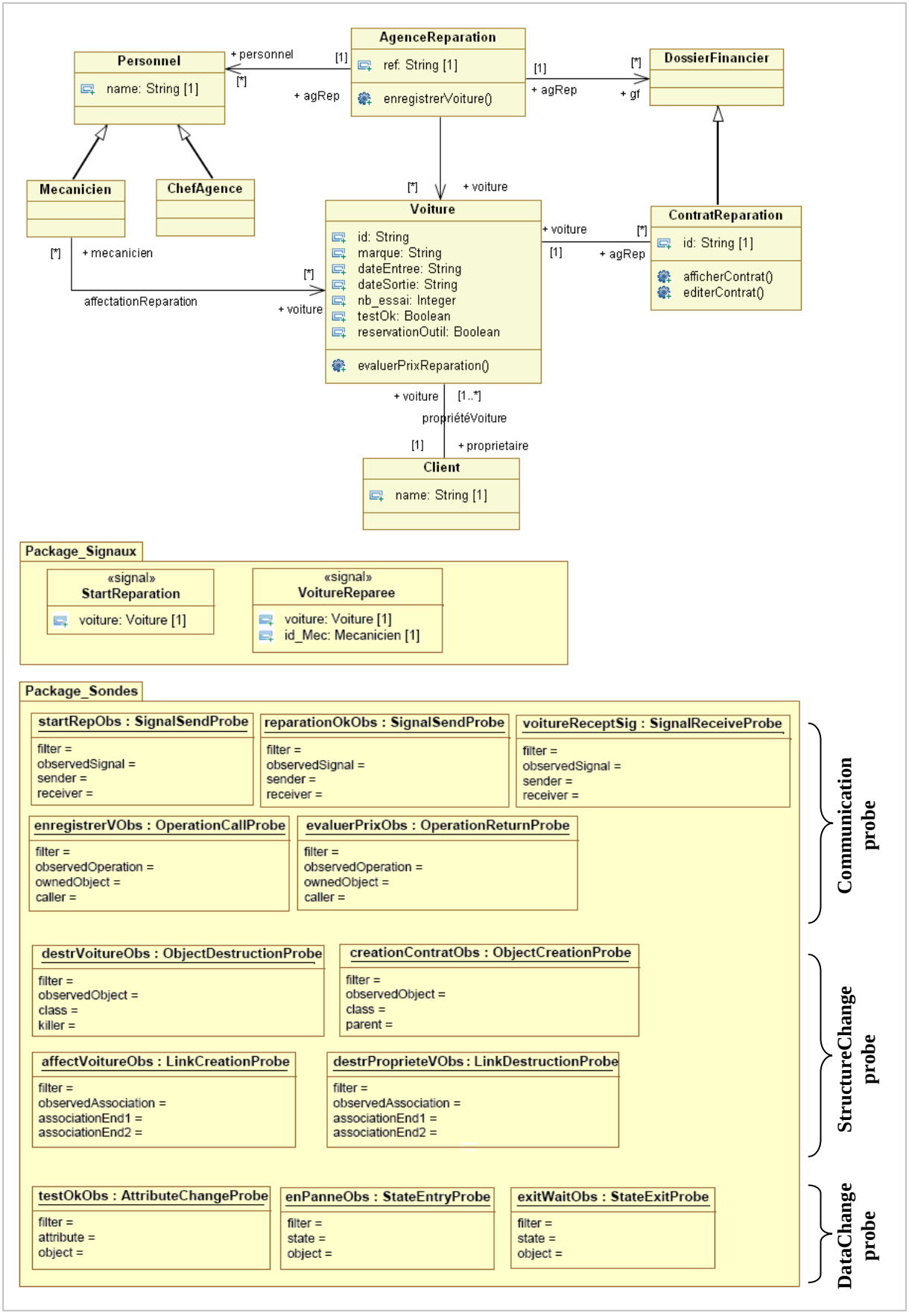


Figure IV.12. Diagramme de classes simplifié de l'application "Gestion d'une agence de réparation de voitures"

► **Exemples de filtres associés à des sondes de type *SignalProbe* :**

- Sonde qui détecte toute émission de signal de type *startReparation* par le chef de l'agence (cf. Figure IV.13-a).
- Sonde qui contrôle toute émission de signal de type *voitureReparee* par un mécanicien à un responsable d'unité (cf. Figure IV.13-b).
- Sonde qui contrôle toute émission de signal de type *voitureReparee* par le mécanicien portant le nom de *Patrick* à un responsable d'unité (cf. Figure IV.13-c).
- Sonde qui détecte toutes les réceptions de signaux (quels que soient leurs types) par les objets de type *Voiture* (cf. Figure IV.13-d).

(a)
```
startRepObs.filter = " Context startRepObs : SignalSendProbe inv :
                       self.observedSignal.oclIsTypeOf(startReparation)
                       and self.sender.oclIsTypeOf(ChefAgence)
                     "
```

(b)
```
reparationOkObs.filter = " Context reparationOkObs : SignalSendProbe inv :
                           self.observedSignal oclIsTypeOf(voitureReparee)
                           and  self.sender.oclIsTypeOf(Mecanicien)
                           and self.receiver.oclIsTypeOf(ResponsableAtelier)
                         "
```

(c)
```
reparationOkObs.filter = " Context reparationOkObs : SignalSendProbe inv :
                           self.observedSignal oclIsTypeOf(voitureReparee)
                           and    if self.sender.oclIsTypeOf(Mecanicien)
                                  then let m:Mecanicien = self.sender.oclAsType(Mecanicien)
                                       in m.name='Patrick'
                                  endif
                           and self.receiver.oclIsTypeOf(ResponsableAtelier)
                         "
```

(d)
```
voitureReceptSig.filter = " Context voitureReceptSig : SignalReceiveProbe inv :
                            self.receiver.oclIsTypeOf(Voiture)
                          "
```

Figure IV.13. Exemples de filtres sur des sondes de type SignalProbe

► **Exemples de filtres associés à des sondes de type OperationProbe :**

- Sonde détectant tous les appels de l'opération *enregistrerVoiture()* de la classe "AgenceReparation" (Figure IV.14-a),
- Sonde détectant tout retour de l'opération "evaluerPrix" de la classe *Voiture* (cf. Figure IV.14-b).

(a) enregistrerVObs.filter = " Context **enregistrerVObs : OperationCallProbe** inv :
self.**ownedObject**.oclIsTypeOf(**AgenceReparation**)
and self.**observedOperation**.name='**enregistrerVoiture**'
"

(b) evaluerPrixObs.filter = " Context **evaluerPrixObs : OperationReturnProbe** inv :
self.**ownedObject**.oclIsTypeOf(**Voiture**)
self.**observedOperation**.name='**evaluerPrixReparation**'
"

Figure IV.14. Exemples de filtres sur des sondes de type OperationProbe

► **Exemples de filtres associés à des sondes de type ObjectLifeProbe :**

- Sonde pour détecter toute création d'instance de la classe *Contrat* (cf. Figure IV.15-a).
- Sonde détectant la destruction de tout objet de type *Voiture* par l'agence de réparation (cf. Figure IV.15-b).

(a) creationContratObs.filter = " Context **creationContratObs : ObjectCreationProbe** inv :
self.**class**.name='**Contrat**'
"

(b) destrVoitureObs.filter = " Context **destrVoitureObs : ObjectDestructionProbe** inv :
self.**observedObject**.oclIsTypeOf(**Voiture**)
and self.**killer**.oclIsTypeOf(**AgenceReparation**)
"

Figure IV.15. Exemples de filtres sur des sondes de type ObjectLifeProbe

► **Exemples de filtres associés à des sondes de type LinkProbe :**

- Sonde détectant les affectations des voitures aux mécaniciens, par la création de liaisons entre des objets de type *Mecanicien* et *Voiture* (cf. Figure IV.16-a),
- Sonde qui détecte la destruction de la liaison *propriétaire* entre des objets de type *Client* et *Voiture* (cf. Figure IV.16-b).

```
(a)  affectVoitureObs.filter = " Context affectVoitureObs : LinkCreationProbe inv :
                                   self.observedAssociation.name='affectationVoiture'
                                   and self.associationEnd1.oclIsTypeOf(Mecanicien)
                                   and self.associationEnd2.oclIsTypeOf(Voiture)
                               "

(b)  destrProprieteVObs.filter = " Context destrProprieteVObs : LinkDestructionProbe inv :
                                   self.observedAssociation.name='propriétéVoiture'
                                   and self.associationEnd1.oclIsTypeOf(Client)
                                   and self.associationEnd2.oclIsTypeOf(Voiture)
                                 "
```

Figure IV.16. Exemples de filtres sur des sondes de type LinkProbe

► **Exemples de filtres associés à des sondes de type AttributeChangeProbe :**

- Sonde qui détecte le passage de l'attribut *testOK* à *true* des objets de type *Voiture* (cf. Figure IV.17).

```
testOkObs.filter = "   Context testOkObs : AttributeChangeProbe inv :
                           self.object.oclIsTypeOf(Voiture)
                           and self.attribute.name='testOk'
                           and (let m:Voiture = self.object.oclAsType(Voiture)
                           in m.testOk=true)
                   "
```

Figure IV.17. Exemple de filtres sur des sondes de type AttributeChangeProbe

► **Exemple de filtres associés à des sondes de type StateChangeProbe :**

- Sonde qui détecte l'entrée dans l'état *EnPanne* des objets *Voiture* (cf. Figure IV.18-a).
- Sonde détectant toute sortie de l'état *EnAttente* des objets *Voiture* (cf. Figure IV.18-b).

```
(a)  enPanneObs.filter = " Context enPanneObs : StateEntryProbe inv :
                             self.object.oclIsTypeOf(Voiture)
                             and self.state.name='EnPanne'
                           "

(b)  exitWaitObs.filter = " Context exitWaitObs : StateExitProbe inv :
                             self.object.oclIsTypeOf(Voiture)
                             and self.state.name='EnAttente'
                           "
```

Figure IV.18. Exemples de filtres sur des sondes de type StateProbe

IV.3.2.5. Dérivation des sondes

Les sondes de la bibliothèque *ProbeLibrary* contiennent les méta-données de l'événement observé par la sonde. Le concepteur peut déclarer de nouveaux attributs pour stocker des informations supplémentaires sur l'état du système au moment de l'activation de la sonde. Cela est réalisé en dérivant le type de sonde considéré.

Considérons une sonde de type *SignalSendProbe*, qui sera utilisée pour la détection du signal *testOK* envoyé par les objets de type *Voiture*. Soit une instance de la sonde *SignalSendProbe* avec un filtre adéquat pour réaliser cette observation. Si on veut – au moment de l'activation de la sonde – sauvegarder l'heure de l'envoi du signal *testOK*, on doit étendre cette sonde par un attribut supplémentaire, qu'on note *date* par exemple. La Figure IV.19 ci-dessous montre la classe de la sonde dérivée *SignalSendProbe_testFonction* (Figure IV.19-a) ainsi qu'une instance de cette dernière avec son filtre (Figure IV.19-b)

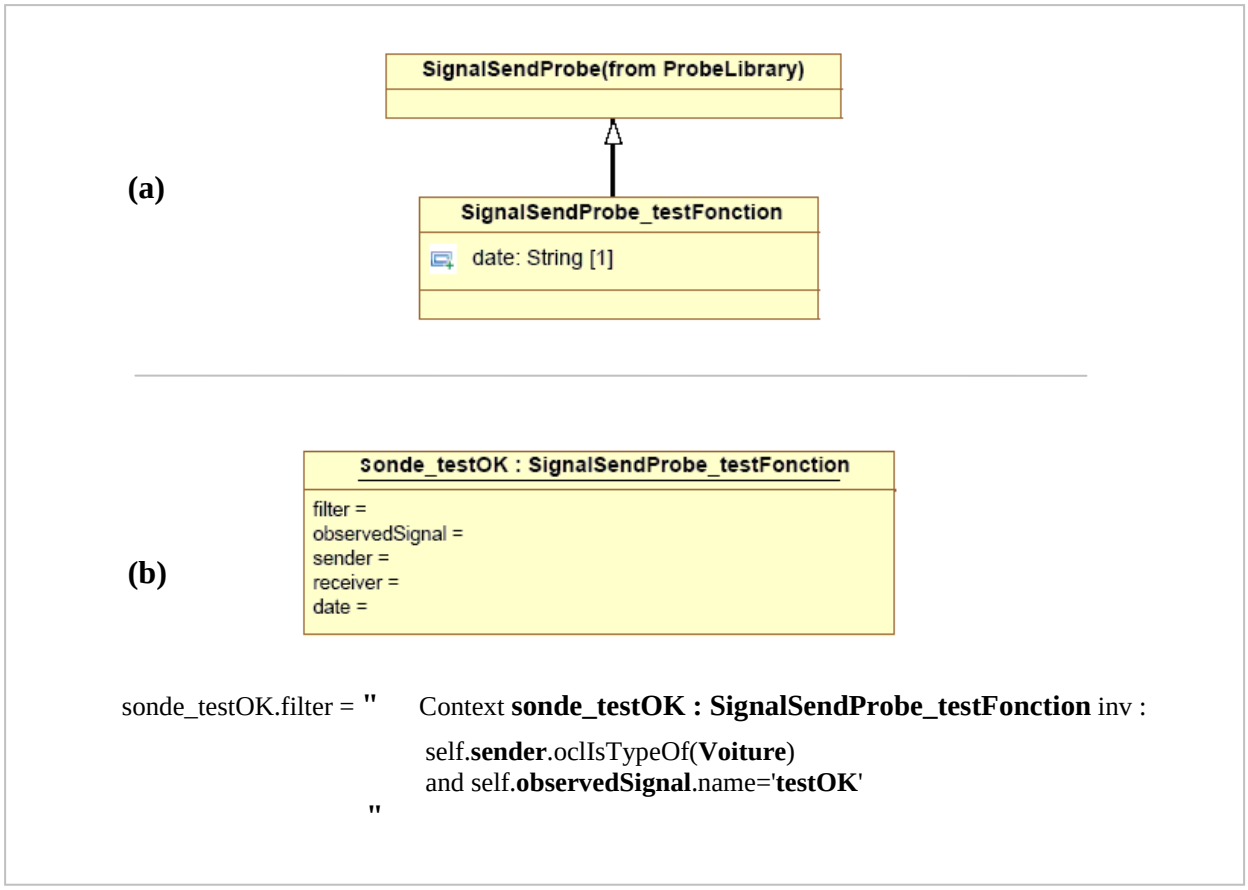


Figure IV.19. Exemple de dérivation de la sonde SignalSendProbe

Nous rappelons que l'opération *update()* d'une sonde est l'opération responsable de la mise à jour des attributs de la sonde lors de son activation. En conséquence, cette opération doit être redéfinie dans la classe dérivée pour pouvoir mettre à jour les nouveaux attributs rajoutés (*date* dans notre exemple). Le code suivant, écrit en Java, donne le squelette de l'opération *update()* de la classe *SignalSendProbe_testFonction*.

```
Void update()
  {
      super.update();
      this.date = this.getSysDate(); //getSysDate() méthode membre qui
                                     //retourne la date du système.
  }
```

IV.3.2.6. Composition de sondes élémentaires

Nous avons abordé jusqu'à présent deux mécanismes capables de personnaliser les sondes élémentaires. Premièrement la projection de sonde, basée sur un filtre, qui offre la possibilité de personnaliser une sonde à un contexte particulier en utilisant les données et les méta-données des événements recherchés (cf. section IV.3.2.4). Deuxièmement la dérivation de sonde, qui offre la possibilité de rajouter de nouveaux attributs pour stocker des informations supplémentaires sur l'état du système lors de l'activation de la sonde (cf. section IV.3.2.5). Néanmoins, ces deux mécanismes ne permettent de personnaliser qu'un seul type de sonde élémentaire à la fois, et par conséquent de réaliser des sondes qui n'agissent que sur un seul type d'événements. Les deux mécanismes ne permettent pas de représenter des sondes plus complexes basées sur plus d'un type d'événement. Dans cette section nous présentons un troisième mécanisme, la composition de sondes, dont l'objectif est de pouvoir observer plusieurs types d'événements au même moment.

IV.3.2.6.1. Définition et principe de composition

Nous avons choisi de représenter la composition de sondes par une machine à états. Ce choix est motivé par plusieurs raisons, parmi lesquelles on peut citer : (1) simplifier le travail des concepteurs pour exprimer les formules de composition, c'est-à-dire exprimer la composition sous forme graphique et non sous forme de formules mathématiques ; (2) donner la possibilité de représenter l'ordre temporel d'apparition des événements dans le système ; (3) d'une manière générale, bénéficier des moyens et des facilités qu'offre une machine à états concernant la description des comportements sous forme d'états/transitions.

Prenons par exemple la situation simple où l'on veut référencer les moments de manifestation (ou d'apparition) du comportement A suivi du comportement B. On suppose que le comportement A est observé par la sonde obs1, et le comportement B observé par la sonde obs2. La Figure IV.20-a illustre la composition des deux sondes obs1 et obs2 sous la forme d'une machine à états.

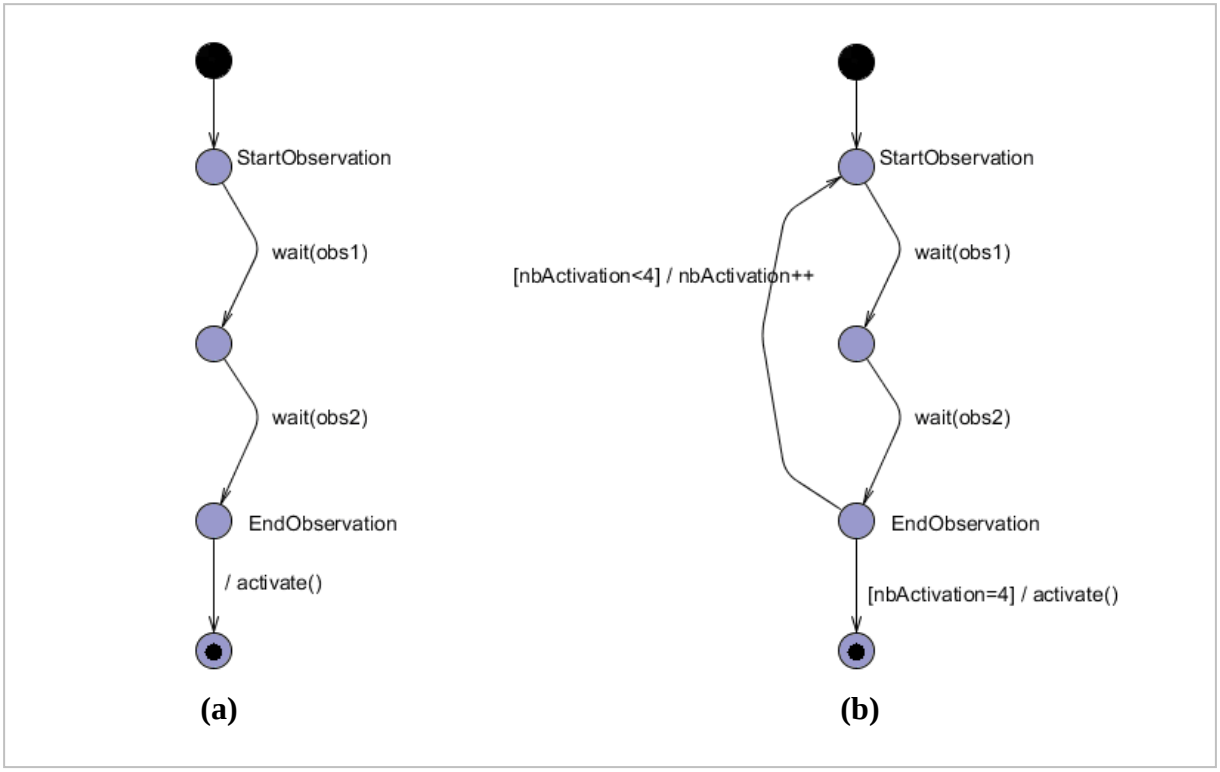


Figure IV.20. Exemple de composition de sondes

Sur le même exemple, si nous voulons maintenant que la sonde ne soit déclenchée qu'après quatre réalisations de la séquence (comportement A, comportement B), la machine à états représentant cette composition est donnée par la Figure IV.20-b, où la variable *nbActivation*, initialisée à 1, compte le nombre de réalisations de la séquence. Comme on peut le voir sur les deux machines, entre les deux états *startObervation* et *EndObservation*, on peut placer tout scénario lié à la définition de la sonde composée. A la sortie de l'état *EndObservation*, l'opération *activate()* est exécutée, déclarant ainsi la réalisation du comportement observé.

Dans le cas des sondes élémentaires, l'opération *projection()* est l'opération responsable de la sélection des événements qui répondent aux contraintes édictées par le filtre de la sonde (l'attribut *filter*). Dans le cas des sondes composées – pour garder la cohérence dans notre démarche – l'opération *projection()* conserve le même rôle, elle est responsable de la vérification de la réalisation du/des comportement(s) observé(s). Cela est fait en associant la machine à états représentant la composition des sondes à l'opération *projection()*.

Pour résumer, le terme de projection utilisé ici désigne l'opération qui spécifie le contexte d'une sonde. La projection est exprimée de deux manières : (i) par le filtre (représenté par l'attribut *filter*) dans le cas des sondes élémentaires, (ii) par des règles de composition (représentées par la machine à états associée) dans le cas des sondes composées.

IV.3.2.6.2. Déclaration d'une classe de sondes composées

La déclaration des sondes composées se fait par dérivation de la classe racine *Probe* de la bibliothèque *ProbeLibrary*. Pour déclarer une sonde composée, il faut suivre la démarche suivante :

(1) dériver la classe racine *Probe* de la bibliothèque *ProbeLibrary* ;

(2) donner à l'attribut *isComposite* la valeur *true* ;

(3) ajouter des attributs supplémentaires en cas de besoin ;

(4) associer la machine à états représentant la composition à l'opération *projection()*.

IV.3.3. Intégration de la notion de sondes dans UML

La notion de sonde que nous présentons est orthogonale à tout autre élément proposé dans UML. En effet, les objectifs visés avec la notion de sonde d'événements ne peuvent pas se concrétiser en utilisant les éléments de modélisation existants. Cette présentation des sondes dans UML est faite dans l'optique de réaliser un profil UML pour manipuler et valider les concepts introduits (cf. section IV.3.3.2). Dans cette sous-section nous montrons comment utiliser les sondes dans une modélisation UML.

IV.3.3.1. Généralités sur les profils UML

L'un des points forts d'UML est qu'il est général et non dédié à un domaine spécifique. Pour pouvoir spécialiser UML pour un usage spécifique, son méta-modèle a la capacité d'être étendu pour intégrer de nouvelles constructions et de nouvelles règles d'assemblage adaptées au domaine traité. Cette opération a un double avantage : d'abord elle permet de prendre en compte les exigences d'un domaine d'application particulier en proposant un cadre de modélisation adapté à ce domaine ; ensuite, elle permet d'utiliser les outils classiques propres à UML pour les appliquer au profil défini et ainsi de bénéficier des traitements spécifiques au domaine. Le *profil* est le mécanisme standard d'extension d'UML. Il a été introduit dans le standard UML depuis la version 1.3 comme un moyen permettant la structuration des extensions UML.

Les profils UML sont conformes au méta-méta-modèle MOF de l'OMG. Les profils UML combinent les concepts de stéréotype, de valeur marquée et de contrainte afin de fournir un nouveau langage compatible avec UML pour un domaine spécifique d'application. Un stéréotype fournit des éléments ayant des rôles spécifiques et permet le stockage d'informations supplémentaires, spécifiques du contexte, dans les valeurs marquées. Les profils permettent aussi de définir des contraintes qui garantissent l'intégrité d'un système en définissant des propriétés sémantiques.

Dans ce contexte, l'objectif de cette section est de structurer notre approche à base de sonde d'événements – qui propose une extension du méta-modèle UML – sous forme d'un profil UML. Nous développons les éléments de modélisation de niveau M2 nécessaires pour supporter les concepts introduits dans ce chapitre.

IV.3.3.2. Probe_profile : profil UML pour représenter les sondes

Les travaux réalisés dans VUML ont traité les aspects structurels de l'approche par points de vue, tels que la notion de classe multivue, de base, de vue, et l'association de dépendance *viewExtension*. L'objectif de cette section est de présenter la syntaxe du profil *Probe_profile* (complémentaire du profil VUML) ainsi que sa sémantique statique.

IV.3.3.2.1. Méta-modèle pour les sondes d'événements

Nous avons défini la syntaxe abstraite et la sémantique statique du profil *Probe_profile* sous forme d'un méta-modèle décrivant les éléments pertinents introduits, et de règles de bonne formation. Ce méta-modèle décrit les concepts liés à la notion de sonde d'événements conformément au MOF, puis traduit vers un profil UML. Il introduit un certain nombre d'éléments de modélisation, à savoir : *Probe*, *ProbeClass*, *ProbeUse*, *ProbeEvent*, et *Wait*. La Figure IV.21 suivante donne une vue générale du méta-modèle proposé.

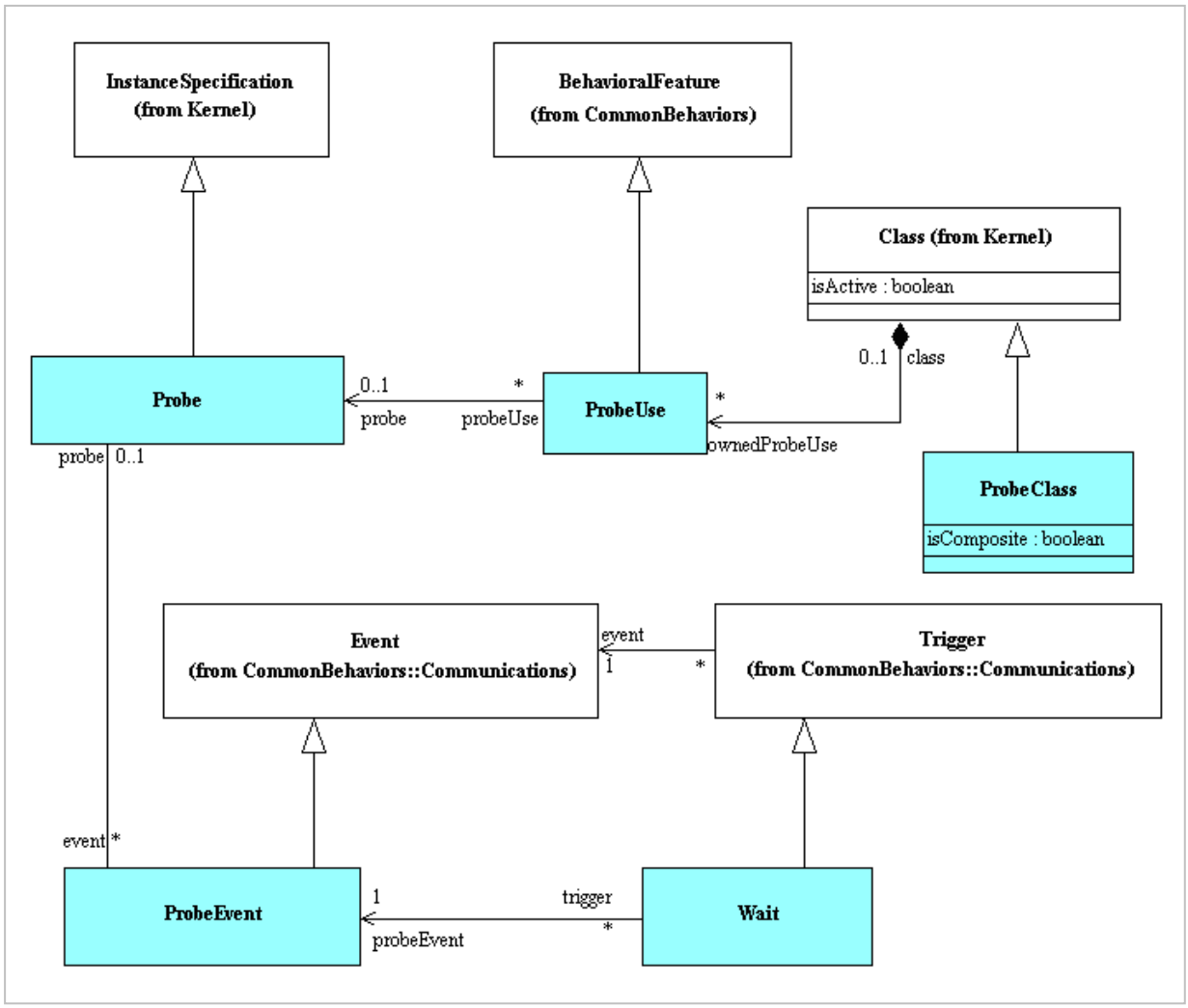


Figure IV.21. Vision générale du méta-modèle VxUML proposé

IV.3.3.2.1.1. Méta-classe ProbeClass

ProbeClass est une méta-classe qui représente les types de sondes prédéfinis regroupées dans la bibliothèque *ProbeLibrary*. Nous l'avons défini pour unifier les propriétés communes aux différentes familles de sonde. Une classe de sondes est un *Classifier* UML susceptible d'avoir des attributs, des méthodes, et qui peut également avoir un comportement représenté par des opérations et des machines à états. Nous la modélisons comme une sous-classe de la méta-classe *Class*. Une sonde est soit élémentaire, soit composée.

Attributs

- isComposite : Boolean

  Attribut booléen ayant pour objectif de différencier les sondes élémentaires des sondes composées. La valeur par défaut est *false* signifiant que la sonde est élémentaire.

IV.3.3.2.1.2. Méta-classe Probe

Une sonde est un élément de modélisation fournissant un mécanisme concret pour la spécification du comportement. Une sonde est présentée dans le modèle de conception sous forme d'une instance d'une des classes de sondes. A cet effet, nous la définissons dans le méta-modèle comme étant une sous-classe de la méta-classe *InstanceSpecification*.

Associations

- event : Event[*]
  Un événement de type *ProbeEvent* désigne l'activation d'une sonde et indiquant ainsi la réalisation de l'observation. La cardinalité * signifie qu'une sonde peut être activée plusieurs fois.
- probeUse : ProbeUse[*]
  Une sonde peut être référencée et utilisée par plusieurs *probeUse*

IV.3.3.2.1.3. Méta-classe ProbeUse

La déclaration d'une sonde est faite d'une manière indépendante d'une autre entité du système. Pour qu'une classe puisse utiliser une ou plusieurs sondes, elle doit déclarer des références pour chacune de ces sondes. Nous faisons ici l'analogie avec la déclaration d'une réception pour chacun des signaux qu'une classe va éventuellement utiliser. Cette référence est modélisée par un élément de type *ProbeUse* (détaillé dans la section IV.3.3.2.3) qui spécialise la méta-classe *BehavioralFeature* d'UML.

Associations

- class : Class[0..1]
  Association qui référence la classe possédant la *probeUse*.
- ownedProbeUse : ProbeUse[*] (du coté de la méta-classe *Class*)

  Indique qu'une classe peut déclarer plusieurs *probeUse*.

- probe : Probe[0..1]
  Une probeUse peut faire référence à au plus une sonde.

IV.3.3.2.1.4. Méta-classe ProbeEvent

L'information sur l'activation de la sonde dans le système est exprimée par la génération d'un nouveau type d'événement UML, qui est *ProbeEvent*. Comme les autres événements UML, le trigger dans une transition relie le mécanisme de communication déclencheur avec l'instance dont on spécifie le comportement à travers l'événement associé au mécanisme, tel que *CallEvent* ou *SignalEvent*. Ici le trigger *Wait* fait référence au mécanisme de sonde à travers son événement associé *ProbeEvent*.

ProbeEvent est une méta-classe qui représente les événements désignant les sondes lors de leur activation. Nous définissons cette méta-classe comme une sous-classe de Event.

Associations

- probe : Probe[0..1]
  Association qui associe la sonde avec l'événement *ProbeEvent*. La cardinalité * du coté *ProbeEvent* signifie qu'une même sonde peut être activée plusieurs fois, générant à chaque fois un événement de type *ProbeEvent*.
- Trigger : Wait[*]
  La cardinalité * signifie que plusieurs triggers de type *Wait* peuvent faire référence à une même activation de sonde (cardinalité 1 du coté *ProbeEvent*).

IV.3.3.2.1.5. Méta-classe Wait

L'utilisation des sondes par une classe se fait au niveau de sa machine à états à travers le type de trigger *Wait* introduit à cet effet. Ce trigger opère sur le type d'événement *ProbeEvent* qui représente l'activation d'une sonde. La méta-classe *Wait* est définie comme une sous-classe de la méta-classe Trigger.

Associations

- *ProbeEvent* : *ProbeEvent*[1]
  Association qui spécifie l'événement attendu par le trigger. La cardinalité 1 du coté *ProbeUse* signifie qu'un trigger de type *Wait* ne peut référencer qu'un seul événement de type *ProbeEvent* à la fois.

IV.3.3.2.2. Stéréotypes du profil Probe_profile

Le profil VxUML étend le méta-modèle UML en introduisant un certain nombre de stéréotypes : *ProbeClass*, *Probe*, *ProbeUse*, *ProbeEvent*, et *Wait*. Ces stéréotypes sont le résultat de l'opération de mapping à partir du méta-modèle présenté dans la section précédente (cf. IV.3.3.2.1). Ces éléments permettent d'établir une correspondance entre les concepts UML et les concepts du

domaine représenté par notre profil VxUML. Nous présentons ici nos choix concernant la traduction en stéréotypes des éléments du méta-modèle proposé :

- le stéréotype ***« probeClass »*** étend la méta-classe UML *Clas*. Il sert à représenter les types de sondes de la bibliothèque *ProbeLibrary*. Il sert également à faire la différence entre les deux catégories de sondes (composite et élémentaire) à travers la valeur marquée *isComposite*. La valeur *false* (attribuée par défaut aux nouveaux éléments créés) dénote que la classe de sondes est élémentaire.

- le stéréotype ***« probe »*** étend la méta-classe UML *InstanceSpecification*. Il sert à dénoter les objets instances des classes stéréotypées *« probeClass »*. Pour assurer que ce stéréotype ne soit utilisé que sur des instances des classes de la bibiothèque, nous avons développé une contrainte OCL et l'avons associé au profil (cf. section suivante).

- le stéréotype ***« probeUse »*** étend la méta-classe UML *Reception*. Une réception dans UML est un élément de modélisation qui annonce l'utilisation d'un signal par la classe qui a déclaré la réception. Nous avons étendu cette méta-classe car elle est la plus proche dans son concept de la sémantique de *ProbeUse*. Avec ce choix, nous avons évité de devoir étendre la méta-classe *Class* pour créer un nouveau stéréotype représentant les classes utilisatrices des sondes.
  Au niveau syntaxique, référencer une sonde obs1 au sein d'une classe se fait en déclarant une réception de signaux stéréotypée par *« probeUse »* de la manière suivante : *« probeUse »* obs1.

- le stéréotype ***« probeEvent »*** étend la méta-classe UML *AnyReceiveEvent*. Notons que la méta-classe *Event* est une méta-classe abstraite non extensible.

- le stéréotype ***« wait »*** étend la méta-classe UML Trigger. La syntaxe *wait(obs)* utilisée dans une transition spécifie que le trigger *wait* attend l'activation de la sonde *"obs"* pour déclencher cette transition.

La Figure IV.22 résume la structure statique du profil *Probe_profile*.

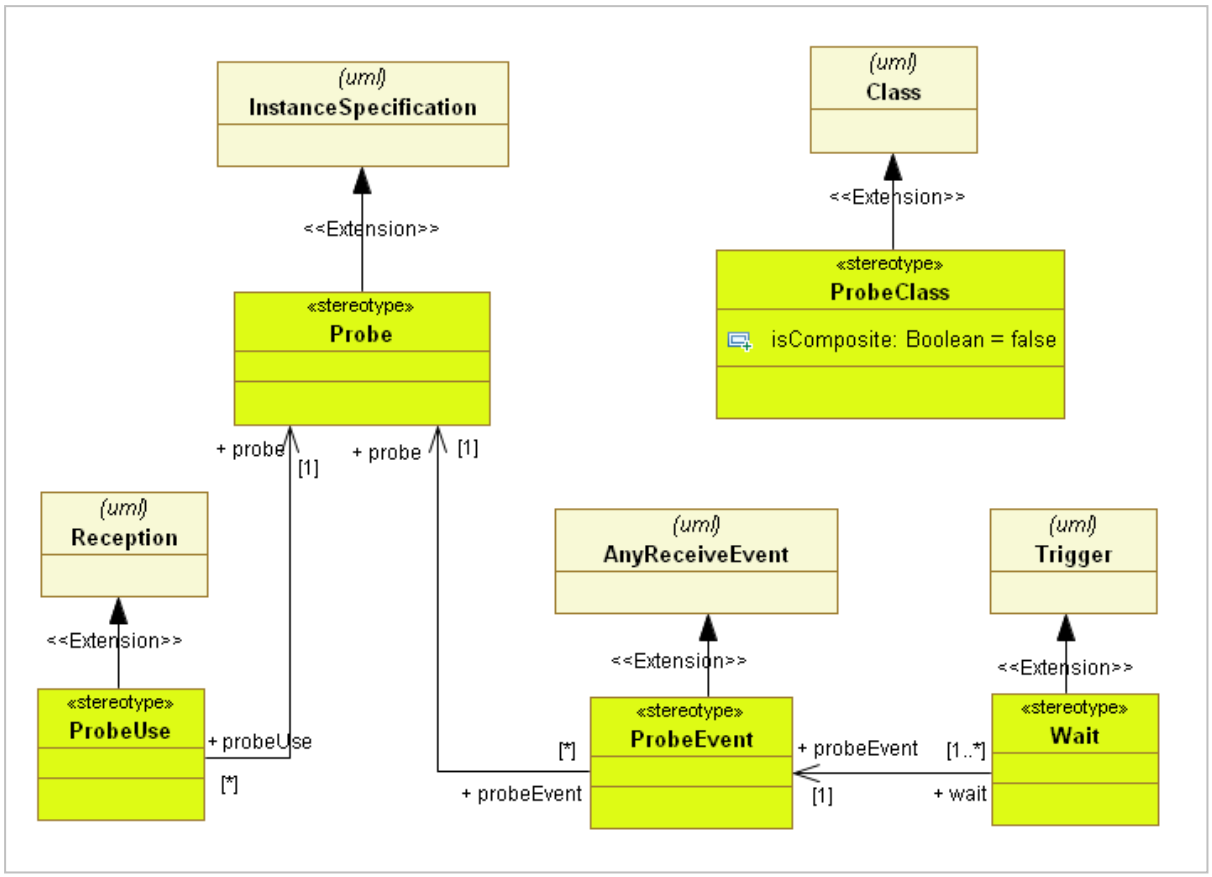


Figure IV.22. Définition des stéréotypes du profil Probe_profile

IV.3.3.2.3. Règles de bonne formation associées au profil Probe_profile

Nous donnons ici quelques règles sémantiques associées aux éléments définis dans le profil. Elles sont décrites en langage naturel. Leur traduction dans le langage OCL est présentée dans la section VI.3.2.2 dédiée à l'implémentation du module de vérification sémantique du prototype VxUML.

[1]. Un élément Class stéréotypé par « probeClass » ne peut pas participer dans des associations.

[2]. Si un élément Class stéréotypé par « probeClass » participe dans une relation d'héritage, le deuxième élément Class participant à cette relation doit être également stéréotypé par « probeClass ».

[3]. Une classe de sondes composées doit posséder une machine à états représentant les règles de composition.

[4]. La classe doit être active pour pouvoir utiliser les sondes.

[5]. Une classe doit déclarer une réception « probeUse » pour chaque sonde utilisée.

[6]. Une *probeUse* peut être utilisée pour tous les sous-types de la sonde référencée par cette *probeUse*.

[7]. Si une *InstanceSpecification* stéréotypée par « probe » possède un classifieur, ce dernier doit être une classe stéréotypée par « probeClass ».

[8]. Si un élément *Reception* est stéréotypé par « probeUse » il ne peut pas faire référence à un élément de type *Signal*.

[9]. Si un élément *Trigger* est stéréotypé par « wait » il ne peut pas faire référence à un élément de type *Event*.

IV.3.3.3. Utilisation des sondes dans UML

Après avoir présenté le profil Probe_profile, nous illustrons dans cette section les différents stéréotypes introduits sur deux exemples concrets de sondes. Le premier est un exemple simple traitant le cas d'une sonde élémentaire. Le deuxième est un exemple plus complexe manipulant les trois mécanismes de définition de la portée d'une sonde (la projection, la dérivation et la composition de sondes).

IV.3.3.3.1. Exemple d'utilisation des sondes élémentaires

Soit Ma_Classe une classe désirant se servir d'une sonde pour détecter les signaux *startReparation* envoyés par le chef d'agence. La démarche à suivre pour mettre en œuvre l'observation est la suivante :

1. Instancier la classe de sondes assortie au type de l'événement à observer. Il s'agit dans ce cas de la classe *SignalSendProbe*. Nous avons nommé cette instance *startRepObs*. Cela est réalisé en déclarant une InstanceSpecification et puis en la stéréotypant par *« probe »*.
2. Spécifier le filtre de la sonde en décrivant les contraintes à vérifier sur les attributs et méta-attributs associés au type de sonde utilisé. Dans notre exemple, nous avons deux contraintes à appliquer sur les attributs *observedSignal* et *sender*.
3. Déclarer une référence à la sonde *startRepObs* dans la classe utlisatrice de la sonde *Ma_Classe*. Cela est réalisé en déclarant une réception dans la classe et puis en la stéréotypant par *« probeUse »*.
4. Utiliser la sonde dans la spécification du comportement de *Ma_Classe* au sein de sa machine à états.

La Figure IV.23 donne un aperçu général de l'utilisation de la sonde *startRepObs* dans la description du comportement de *Ma_Classe*. La Figure IV.23-a illustre la définition de la sonde *startRepObs*, la Figure IV.23-b présente le référencement de la sonde *startRepObs* par *Ma_Classe*, et la Figure IV.23-c montre l'utilisation de la sonde par la machine à états associée à la classe *Ma_Classe*.

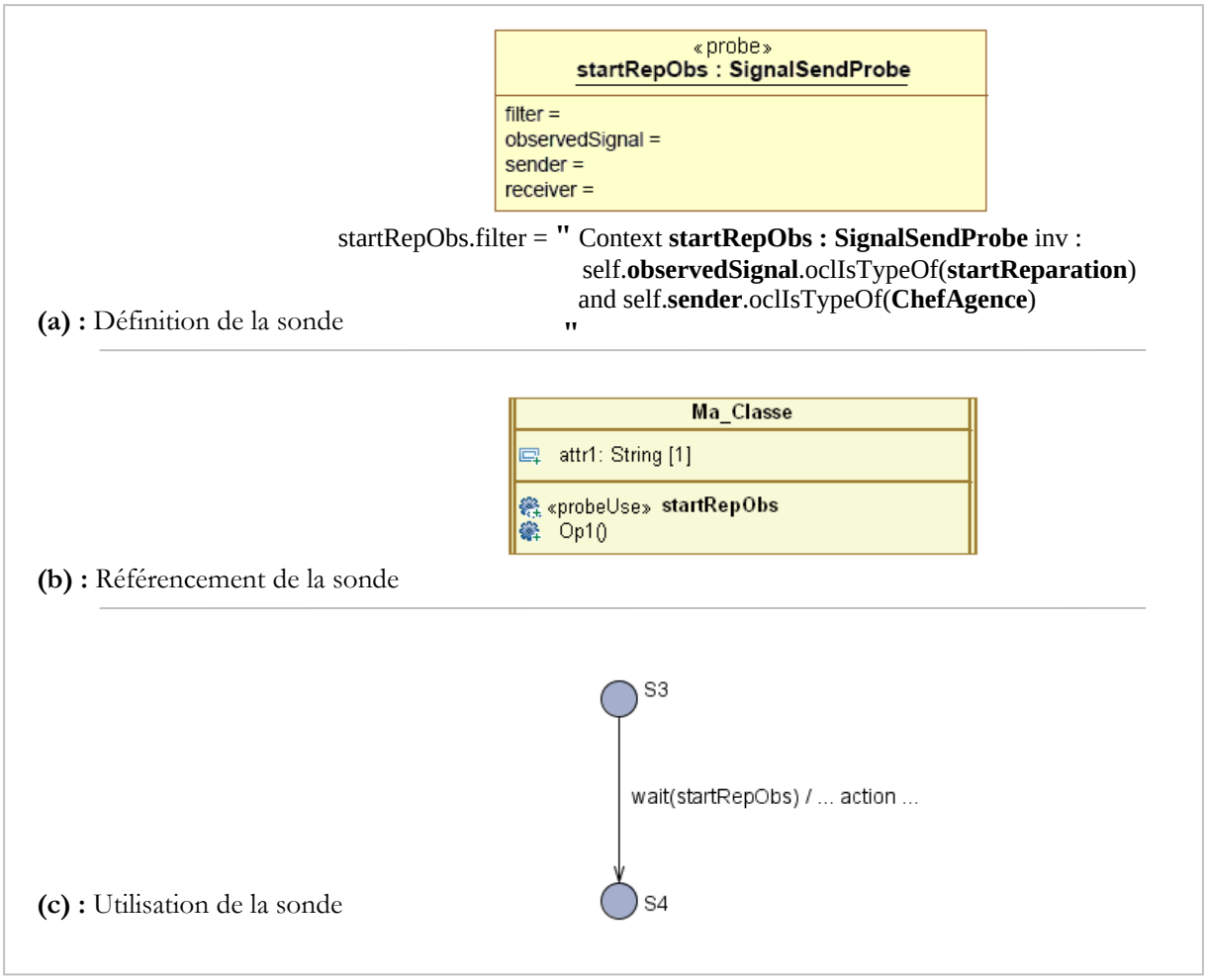


Figure IV.23. Principe général d'utilisation d'une sonde

IV.3.3.3.2. Exemple d'utilisation des sondes composées

Dans cet exemple, nous allons construire la sonde Verif_preReparation, qui est une sonde composée de trois sondes élémentaires. Deux d'entre elles sont de type *SignalSendProbe* et la troisième est de type *AttributeChangeProbe*. Il s'agit d'observer le système pour récolter les informations nécessaires pour pouvoir débuter la réparation d'une voiture. Nous proposons pour cette observation les contraintes suivantes :

- (c1) La permission de réparation d'une voiture peut être donnée soit par le chef d'agence soit par le responsable d'unité.
- (c2) Il faut vérifier si la réservation des outils et des pièces de rechange est faite ou pas.
- (c3) Au moment de la réalisation du comportement, il faut sauvegarder la date de l'émission de l'ordre et le responsable de cet ordre.

Pour répondre à ce besoin, nous déclarons les sondes suivantes :

- **ordre_RA_Rep_Obs.** Cette sonde, de type *SignalSendProbe*, détecte l'envoi du signal ordreRA_Rep_Ok qui désigne l'accord du responsable de l'atelier pour débuter la réparation.
- **ordre_CA_Rep_Obs.** Cette sonde, de type *SignalSendProbe*, détecte l'envoi du signal ordreCA_Rep_Ok qui désigne l'accord du chef de l'agence pour débuter la réparation.
- **reservation_Materiel.** Cette sonde, de type *AttributeChangeProbe*, détecte le changement de l'attribut booléen *reservationOutil* de la classe Voiture quand il passe de la valeur false à true ; il indique que la demande de réservation des outils et des pièces de rechange nécessaires à la réparation est assurée.

La définition des trois sondes élémentaires pré-citées suit la même démarche utilisée dans l'exemple de la sous-section précédente.

- **Verif_preReparation** est la ***nouvelle classe de sondes*** à créer pour composer les trois sondes précitées. Le processus à suivre pour définir cette nouvelle classe de sondes est le suivant :
  - Dériver la classe *Probe* (la classe racine de la bibiothèque *ProbeLibrary*) ; le nom de la classe dérivée est *Verif_preReparation* ;
  - Stéréotyper la classe dérivée par *« classProbe »* et affecter à l'attribut *isComposite* la valeur *true* ;
  - Compléter la classe dérivée par les attributs/opération en cas de besoin. Dans notre exemple, on ajoute deux attributs dateOrdre et responsable pour répondre à la contrainte (c3) ;
  - Déclarer au sein de la classe dérivée des références aux trois sondes élémentaires *ordre_RA_Rep_Obs*, *ordre_CA_Rep_Obs* et *reservation_Materiel* en se servant du stéréotype *« probeUse »* ;
  - Développer la machine à états représentant les conditions de composition des trois sondes et puis l'associer à l'opération *projection()*. Arrivant à ce stade du processus, la classe de sonde *Verif_preReparation* est désormais prête à être instanciée ;
  - Instancier la nouvelle classe. Nous avons nommé cette instance "s".

La Figure IV.24-a illustre la définition des trois sondes élémentaires *ordre_RA_Rep_Obs*, *ordre_CA_Rep_Obs* et *reservation_Materiel*. La Figure IV.24-b illustre la définition et l'instanciation de la classe *Verif_preReparation*, et la Figure IV.24-c illustre la machine à états associée à l'opération *projection()* de cette sonde composée *Verif_preReparation*.

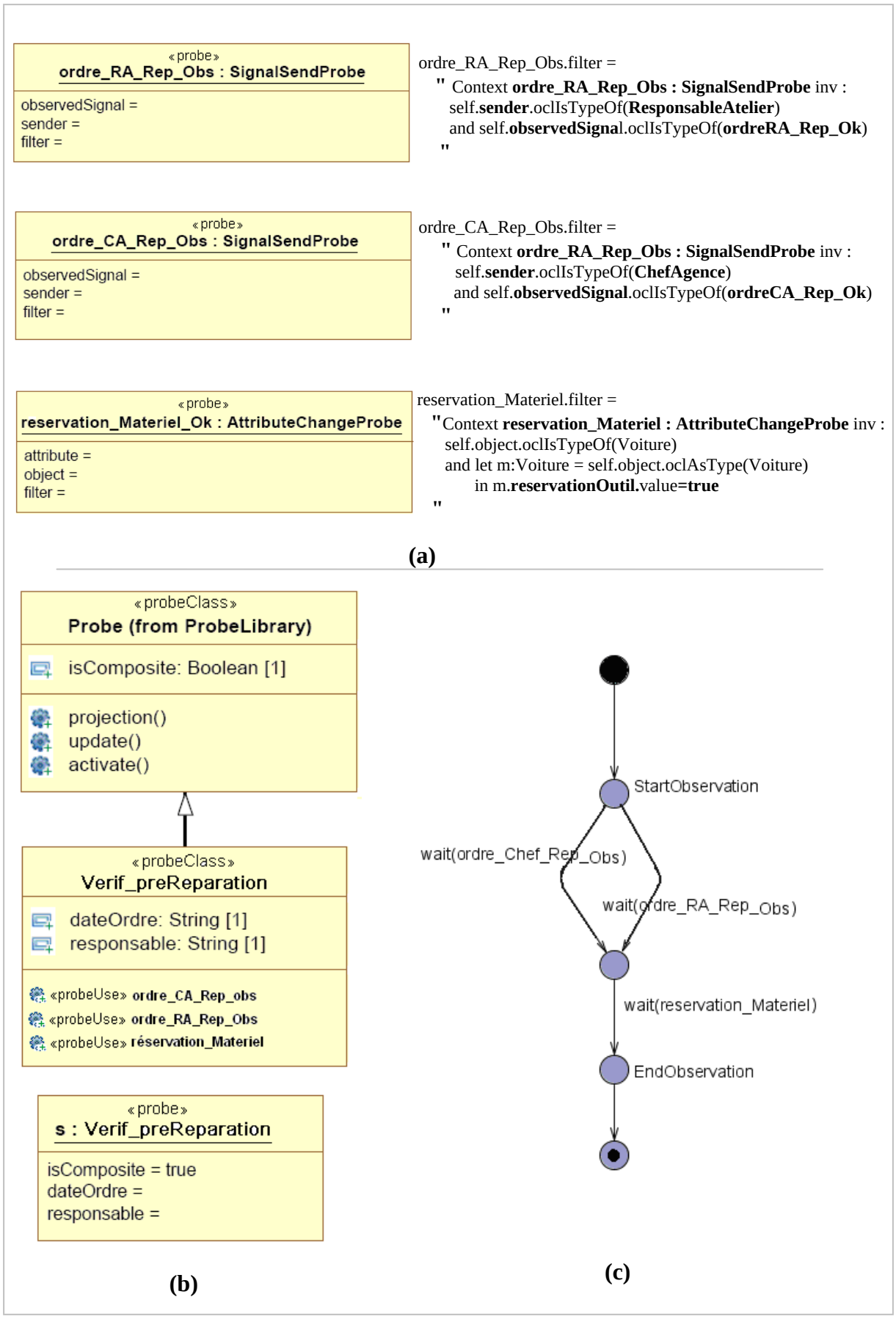


Figure IV.24. Définition de la sonde composée Verif_preReparation

IV.3.4. Spécification et composition du comportement à base de sondes dans VUML

Un des principes de base de la démarche de conception multivue est qu'elle doit assurer au maximum l'indépendance du développement des modèles-vue. Or cette règle peut être violée s'il y a une corrélation dans le développement comportemental des modèles-vue. C'est pour pallier ce problème que nous proposons l'utilisation du concept d'observation basé sur la notion de sondes d'événements.

Si on considère par exemple la machine-vue correspondant au client, celui-ci a besoin de connaître le début réel de la réparation de la voiture. Or cette information n'est pas disponible dans le modèle-vue *Client* car on ne connaît pas dans ce modèle l'acteur responsable de l'action associée au démarrage de la réparation (est-ce une décision du chef de l'agence ? du mécanicien ? du responsable de l'atelier ?) ; on ne connaît pas non plus la manière dont cette action va être représentée (sous forme d'un échange de signal ? de message ? de changement d'attribut ? etc.).

La section suivante présente le principe de la solution apportée par l'utilisation des sondes d'événements pour définir le comportement des vues indépendamment les unes des autres, et présente également le principe de la composition des comportements dans la phase de fusion.

IV.3.4.1. Principe de composition basée sur les sondes d'événements

L'objectif d'introduire la notion de sonde dans VUML est de permettre d'observer des comportements dans le système, de les identifier, et de les utiliser dans la spécification comportementale des objets du système, particulièrement celle des objets-vue. En effet, l'observation d'événements s'avère un moyen efficace capable d'établir une communication implicite entre deux entités du système.

L'utilisation des sondes d'événements dans la démarche VUML se fait de la manière suivante :

- **Déclaration des sondes :** au niveau d'un modèle-vue, deux situations se présentent :
  - ***Le comportement à exprimer peut être accompli en se basant sur les éléments existant dans le modèle-vue.*** Dans ce cas, nous utilisons les mécanismes habituels de communication inter-objets d'UML tels que l'échange de messages et de signaux. Le concepteur peut aussi définir des sondes (élémentaires ou composées) pour observer l'évolution de son modèle. La déclaration et l'utilisation de ces dernières sont similaires à la manière dont les sondes sont présentées dans les sections précédentes. Les sondes ainsi déclarées sont concrètes car toutes les informations nécessaires à leur définition sont disponibles dans le modèle-vue développé.
  - ***Le comportement à exprimer ne peut pas être accompli en se basant uniquement sur les éléments existant dans le modèle-vue.*** Dans ce cas, les conditions de déclenchement d'un comportement ne peuvent être précisées car elles dépendent d'autres points de vue. Pour faire face à cette situation, le développeur peut insérer des sondes sur le comportement recherché. Les sondes dans ce cas ne sont pas complètement décrites car les paramètres du comportement observé ne sont pas définis dans le point de vue développé. Elles ne sont pas concrètes, instanciées

directement de la classe racine *Probe* de la bibliothèque *ProbeLibrary*. En effet, le concepteur en déclarant les sondes abstraites ne se préoccupe pas de la manière dont le comportement recherché va être exprimé ; cette tâche est déléguée aux sondes déclarées. Le type concret de la sonde (une des sous-classes de la classe *Probe*, cf. section IV.3.2.1) sera identifié plus tard dans la phase de fusion quand tous les paramètres de la sonde seront définis. Une note détaillée – expliquant le comportement visé par la sonde – doit être associée à la déclaration de chaque sonde, qui sera utilisée dans la phase de fusion.

- **Utilisation des sondes :** le concepteur utilise les deux genres de sondes déclarées dans l'étape précédente sans préciser si la sonde est abstraite ou non. L'utilisation des sondes dans le modèle comportemental du modèle-vue en développement se fait de la manière habituelle, exposée dans la section précédente (cf. IV.3.3.3)

- **Définition tardive des sondes :** Lors de la phase de fusion, on précise la définition des sondes abstraites déclarées dans l'ensemble des modèles-vue. Le concepteur (ou l'équipe de concepteurs) s'occupant de l'étape de fusion des modèles complète les définitions des différentes sondes abstraites en se basant sur la note UML associée à chaque sonde. Il cherche comment le comportement observé par une sonde est réalisé dans les autres vues, et complète à cet effet sa définition.

A la fin de l'étape de fusion, toutes les sondes abstraites sont concrétisées et leurs paramètres définis. De cette manière, des liens de communication entre les modèles-vue sont tissés implicitement, c'est-à-dire sans qu'une vue ne déploie des voies de communication directes pour notifier les autres vues de la réalisation d'un certain comportement.

Un avantage très important de cette nouvelle méthode de communication est que la composition des modèles comportementaux (les machines à états) se fait implicitement, et donc évite de devoir modifier les modèles-vue. On peut dire qu'une observation est un moyen de communication sans effet de bord sur le modèle observé.

IV.3.4.2. Illustration

Nous illustrons la démarche d'intégration de la notion de sonde d'événements dans la conception par point de vue par des extraits de notre étude de cas "Gestion d'une agence de réparation de réparation de voitures" présentée brièvement dans le chapitre II et qui sera explicitée davantage dans le prochain chapitre sur la démarche VxUML.

Les trois sous-sections suivantes illustrent les étapes de déclaration, d'utilisation, et de définition de la sonde *demarrageReparation* dans la spécification du comportement du point de vue *Client*.

IV.3.4.2.1. Déclaration abstraite des sondes dans la phase de conception décentralisée

Une fois le diagramme de classes – selon un point de vue – réalisé, nous développons les machines à états correspondant aux classes réactives présentes dans le modèle-vue.

Si on considère par exemple la machine-vue correspondant au client, celui-ci a besoin de connaître le début réel de la réparation de la voiture. Or l'événement à référencer n'est pas déterminé – du fait qu'il dépend d'un autre point de vue pas encore forcément modélisé –, et sa détermination ne sera possible que dans la phase de fusion des modèles-vue. En conséquence, on déclare une sonde abstraite *demarrageReparation*. La Figure IV.25 suivante présente la déclaration de la sonde comme une instance directe de la classe *Probe*.

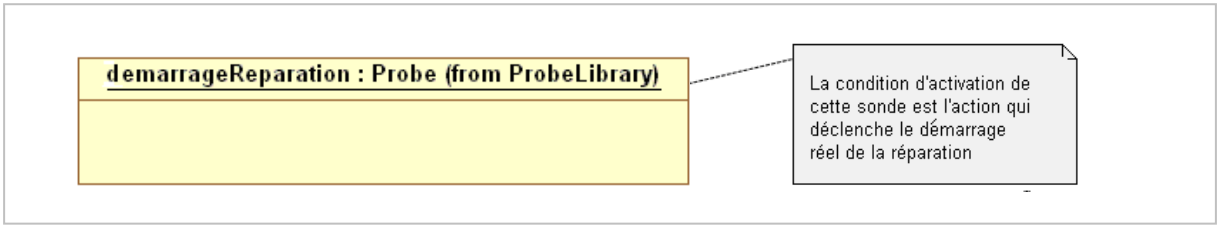


Figure IV.25. Déclaration de la sonde demarrageReparation

IV.3.4.2.2. Utilisation des sondes dans le modèle-vue

La sonde ainsi déclarée peut être utilisée dans la spécification du comportement du point de vue *Client*. Nous utilisons la sonde demarrageReparation dans la vue du client concernant la classe Voiture. Pour pouvoir l'utiliser, il faut déclarer la réception de la sonde dans la classe utilisatrice *Voiture_Client_View* par le biais du stéréotype « *probeUse* » (cf. Figure IV.26-a). La sonde est désormais utilisable dans la spécification du comportement lié à cette classe à travers sa machine à états. Sur la Figure IV.26-b, la sonde *demarrageReparation* représente l'action de déclenchement de la transition entre les deux états *ContratEtabli* et *EnReparation*.

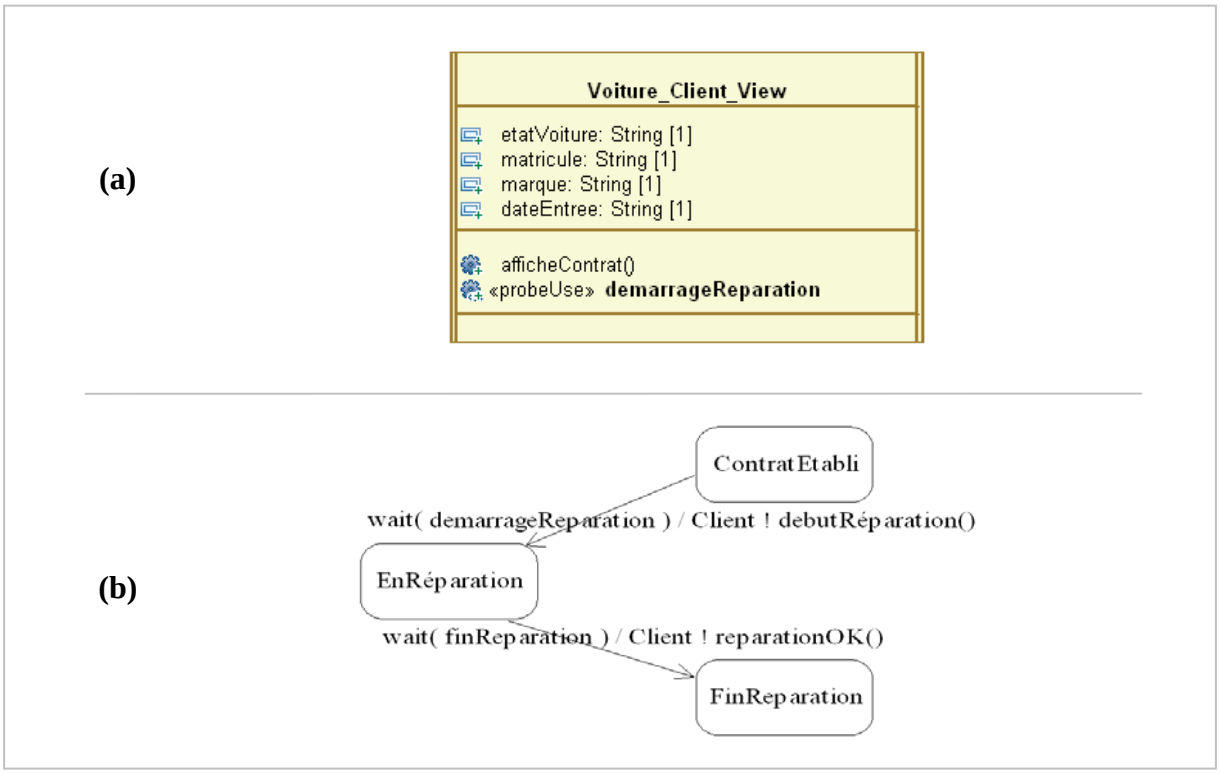


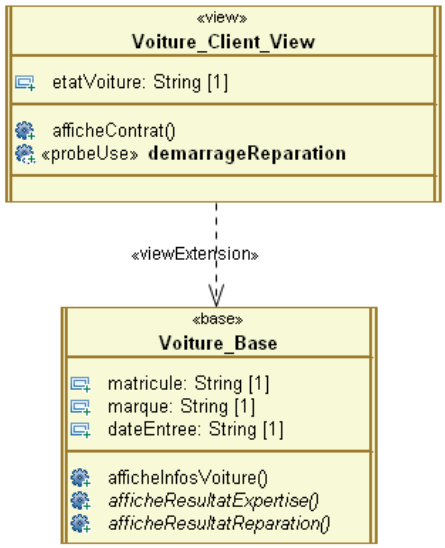
Figure IV.26. Utilisation de la sonde demarrageReparation par la classe Voiture

IV.3.4.2.3. Définition/concrétisation des sondes abstraites dans la phase de fusion

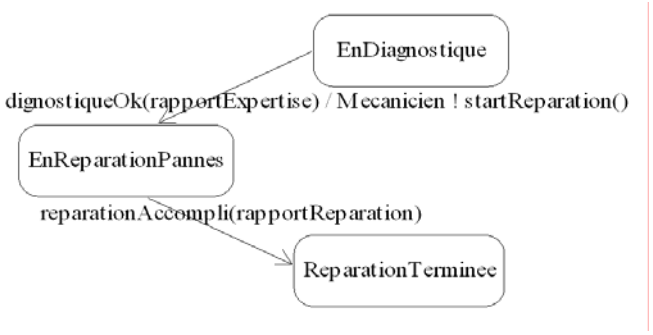
La fusion comportementale – effectuée après la fusion structurelle (cf. chapitre V, sectionV.2) – consiste à décrire le comportement des objets multivue réactifs et à assurer la cohérence de l'ensemble. Un objet multivue est composé d'un ensemble d'objets-vue ayant chacun un comportement décrit par une machine-vue développée lors de la phase de conception par point de vue. La mise en cohérence et les liens entre les machines-vue se fait à travers la concrétisation des sondes qu'elles utilisent, c'est-à-dire la finalisation de leur définition.

La Figure IV.27 illustre le résultat de l'étape de fusion des modèles développés lors de la phase décentralisée. La Figure IV.27-a montre un extrait de la structure de la classe Voiture après la fusion structurelle. Nous n'avons montré que la vue associée à l'acteur client. Une voiture est ainsi représentée par la classe Voiture_Base partagée par l'ensemble des points de vue et par la vue spécifique au point de vue *Client*. La Figure IV.27-c spécifie que le type de la sonde *demarrageReparation* est *SignalSendProbe*, et que le nom du signal est *startReparation* (envoyé au mécanicien, voir la Figure IV.26-b), ce qui permet formellement de faire le lien entre les deux points de vue *Client* et *Mecanicien*.

(a)

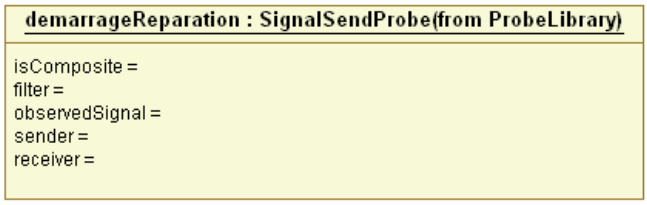


(b)



Fragment de la machine-vue associée à la classe Voiture pour le point de vue *Mecanicien*

(c)



demarrageReparation.filter = " Context **demarrageReparation : SignalSendProbe** inv :
self.**sender**.oclIsTypeOf(**Voiture_Mecanicien_View**)
and self.**receiver**.oclIsTypeOf(**Mecanicien**)
and self.**observedSignal**.name='**startReparation**'
"

Figure IV.27. Utilisation de la sonde demarrageReparation par la classe Voiture

IV.3.5. Bilan de l'approche basée sur les sondes d'événements

Dans cette proposition, nous avons opté pour une approche basée sur l'ajout dans VUML de mécanismes nouveaux, spécifiques pour la modélisation et la composition des comportements des objets-vues. Nous avons introduit pour cela la notion de ***sonde d'événements*** pour spécifier des communications implicites entre les objets-vue à travers des observations d'événements. Ceci permet de découpler des spécifications qui sont a priori fortement interconnectées, de les concevoir séparément, puis de les intégrer sans avoir à les modifier. Pour atteindre notre but nous avons tout d'abord défini le concept de sonde d'événements, identifié les différents types de sondes avec les paramètres associés, puis défini un ensemble de concepts permettant d'enrichir et de manipuler les sondes. Nous avons ensuite proposé une représentation compatible avec VUML des concepts introduits sous forme d'un profil nommé VxUML (extension de VUML). En plus de la définition des éléments du profil (stéréotypes, valeurs marquées, classes de librairie prédéfinies), nous avons développé en OCL des règles de bonne formation (sémantique statique).

IV.4. Conclusion

Dans ce chapitre nous avons traité la problématique de spécification comportementale dans le cadre du profil VUML. Nous nous sommes concentrés sur la description du comportement individuel des objets multivue.

La démarche de conception VUML est une démarche de modélisation orientée point de vue globalement décentralisée, qui procède par des développements partiels de l'application selon les visions subjectives des acteurs du système. Procéder à une modélisation par partie du système offre des avantages, surtout dans le cas des systèmes complexes, mais cette démarche ne permet pas d'assurer l'indépendance dans le développement des vues. En fait, le couplage entre les vues peut s'avérer important si la modélisation du point de vue courant nécessite des informations extérieures à ce dernier. Cette situation rend la démarche de conception par point de vue difficile à mettre en œuvre, voire impossible sans altérer le développement des autres vues afin de recueillir les informations manquantes.

Pour surmonter cette difficulté, il faut traiter les deux points cruciaux suivants : la spécification des comportements des objets-vue d'une part, et la composition de ces comportements pour former le comportement global de l'objet multivue d'autre part. La difficulté du problème réside dans le compromis à proposer une approche permettant une liberté la plus grande possible dans le développement des vues, et en même temps à fournir des moyens pour faciliter la fusion. Pour résoudre ce problème, deux possibilités s'offraient à nous. La première consiste à réutiliser tels quels les mécanismes de spécification du comportement et de communications entre objets d'UML. La deuxième consiste à proposer de nouveaux mécanismes, qui étendent ceux d'UML pour s'adapter aux particularités de VUML.

Analyse de l'approche sans extension d'UML

Cette approche répond à la problématique de la manière suivante : pour favoriser un développement séparé des modèles-vue dans la deuxième phase de la démarche, il est indispensable que cette phase soit guidée par un modèle établi a priori ; d'où la proposition de la notion de machine-base. La machine-base est développée lors de la phase de modélisation globale pour spécifier

le comportement commun aux acteurs. Cette dernière est considérée comme un patron à respecter lors du développement des comportements par points de vue. Dans la phase de modélisation décentralisée, on procède à la description séparée des machines-vue associées aux objets-vue suivant les états composant la machine-base. Dans la phase de fusion, la composition des comportements partiels des différents points de vue est réalisée en ajoutant les échanges de signaux qui permettent la coordination des machines à états des objets-vue et de l'objet-base.

Concernant la mise en œuvre de cette approche, nous avons utilisé le profil Omega UML pour lequel des outils de simulation et de vérification existent [Ober et al., 06]. Toutefois, l'expérience avec cette approche, lors d'un passage à l'échelle montre deux limites : (i) devoir identifier, assez tôt, les classes multivue et leurs machines-base, lors de la phase d'analyse globale. (ii) L'intégration des machines-vue développées séparément peut requérir de nombreuses modifications et adaptations au niveau des machines-vue et de la machine-base.

Analyse de l'approche à base de sonde

Cette approche à base de sonde d'événements résout les deux problèmes évoqués ci-dessus de la manière suivante. Premièrement, elle permet de définir le comportement des vues indépendamment les unes des autres dans la phase de conception décentralisée. Le concepteur n'a pas à se préoccuper, en déclarant des sondes, de la manière dont le comportement recherché va être exprimé, car cette tâche est déléguée aux sondes déclarées. Deuxièmement, l'utilisation des sondes offre un principe simple pour la composition des comportements dans la phase de fusion, et évite de devoir modifier les modèles-vue. En effet, au lieu de modifier les modèles-vue pour réaliser la composition, on procède à une synchronisation de ces derniers en agissant sur les sondes déclarées. Cette mise en cohérence se fait à travers la finalisation de la définition des sondes utilisées dans les différents modèles-vue.

Concernant la mise en œuvre de cette approche, nous avons développé un profil appelé Probe_profile, intégré dans un prototype appelé VxUML (cf. chapitre VI). La critique que nous pouvons apporter à cette approche et le fait que les concepts proposés utilisent à la fois les niveaux de modélisation M1 et M2 ainsi que l'utilisation du langage OCL dans la description des modèles de l'application.

Nous résumons et comparons les deux approches traitées dans ce chapitre dans le tableau suivant :

Points de comparaison	Approche sans extension d'UML	Approche à base de sonde
Concepts propres	Machine-base, machine-vue.	Machine-vue, sonde d'événements, trigger *wait*.
Type d'intégration dans UML	Utilisation des mécanismes UML	Extension d'UML
Principe de spécification du comportement	- La machine-base fournit un patron à respecter par les vues. - Description séparée des machines-vue associées aux objets-vue suit les états composant la machine-base.	- Les modèles-vue sont développés librement. - En cas de couplage entre les vues, on déclare des sondes.
Principe de composition du comportement	Ajout des échanges de messages permettant la coordination des machines à états des objets-vue et de l'objet-base.	Synchronisation des modèles-vue en agissant sur les sondes déclarées
Conformité avec la démarche VUML structurelle	Conforme à VUML	Conforme à VUML
Type d'application visé	Systèmes de petite taille avec une forte indépendance des vues.	Systèmes de moyenne et grande taille avec une forte interdépendance entre les vues. L'approche évite de devoir modifier les modèles-vue dans la phase d'intégration.
Outils support	Utilisation du profil Omega UML	Un prototype VxUML a été développé (cf. chapitre VI)

CHAPITRE V. DEMARCHE VXUML ET APPLICATION A UN CAS D'ETUDE

Sommaire

V.1. Introduction

L'approche de développement par points de vue est une approche centrée acteur, c'est-à-dire que le développement d'une application est subdivisé en plusieurs processus, chacun d'eux construisant le système selon la vision subjective d'un acteur. Le système final est constitué dans une phase ultérieure en fusionnant les différents modèles développés par points de vue (appelés aussi *modèles partiels*). La phase de développement des modèles partiels est une phase effectuée d'une manière décentralisée, où plusieurs équipes peuvent travailler indépendamment les unes des autres. Certes, subdiviser de cette manière la modélisation d'un système en plusieurs modèles constitue un avantage, car cela conduit à une réduction conséquente de la complexité des modèles. Cependant, composer des modèles développés indépendamment les uns des autres est un défi majeur, surtout si on vise à rendre cette composition automatique.

Dans ce cadre de modélisation par points de vue, VUML a été doté d'une démarche composée de trois phases [Nassar 05]. La première est une phase centralisée menant une analyse préalable dans le but de déterminer le périmètre de l'application et constituer un dictionnaire servant de référence pour la deuxième phase. La deuxième phase est une phase décentralisée durant laquelle plusieurs suites de tâches indépendantes sont lancées pour produire les modèles partiels. Chaque équipe de concepteurs, approfondit l'analyse/conception du système pour répondre aux besoins de l'acteur considéré. Enfin, la troisième phase permet de fusionner les modèles-vues, développés dans la deuxième phase, arrivant ainsi au modèle global de l'application, qui est le modèle multivue VUML.

Cependant, la démarche VUML (cf. III.4.4) a été conçue pour aider les concepteurs dans la réalisation des modèles structurels – particulièrement pour aboutir au diagramme de classes – mais sans traiter la problématique de spécification comportementale. Dans le chapitre précédent, nous avons traité ce volet, et nous avons proposé des mécanismes nécessaires à la spécification du comportement dans VUML. Deux approches ont été proposées. La première, basée uniquement sur les concepts d'UML souffrait de limites qui nous ont motivé à adopter une deuxième approche basée sur la notion de sonde d'événements et aboutissant ainsi au profil VxUML.

Nous proposons dans ce chapitre une extension de la démarche associée à VUML pour tenir compte des nouveaux concepts introduits par le mécanisme d'observation, qui est le mécanisme clé de la solution proposée. Cette démarche étend donc la démarche VUML proposée dans [Nassar, 05] et [Anwar, 09].

Afin de montrer l'utilité des concepts introduits dans le chapitre IV, nous proposons dans ce chapitre d'appliquer notre démarche à l'étude de cas traitant le système de gestion d'une agence de réparation de voitures. Il s'agit d'un système d'information complexe, impliquant plusieurs acteurs et se prêtant donc particulièrement bien à une conception par points de vue. A travers cette application, nous visons une modélisation par points de vue la plus complète possible, démontrant ainsi la mise en œuvre de notre approche.

Ce chapitre est composé de deux parties. La première (section 2) présente la démarche VxUML traitant à la fois l'aspect structurel et l'aspect comportemental d'une conception multivue. La deuxième partie (section 3) décrit l'étude de cas détaillée selon la démarche de VxUML. Nous focalisons l'étude sur la spécification du comportement des objets multivue lors de la phase décentralisée et également sur la composition de ce comportement dans la phase de fusion.

V.2. Démarche d'analyse et de conception associée à VxUML

Dans cette section, nous présentons une démarche associée à VxUML qui traite à la fois les aspects structurels et comportementaux. D'une part nous focalisons notre attention sur l'utilisation des concepts comportementaux dans les phases de conception par points de vue et de fusion de modèles, d'autre part nous montrons le rôle des sondes d'événements dans la spécification et la composition du comportement des objets-vue. La Figure V.1 ci-dessous donne une représentation générale de la démarche VxUML. Elle est constituée de trois phases : la phase d'analyse globale, la phase d'analyse/conception par points de vue, et la phase de fusion. Chacune d'elle est constituée de deux volets : le premier guide les développeurs dans la modélisation structurelle d'une application en VUML, le deuxième complète le premier en aidant à exprimer le comportement dynamique des objets multivue identifiés.

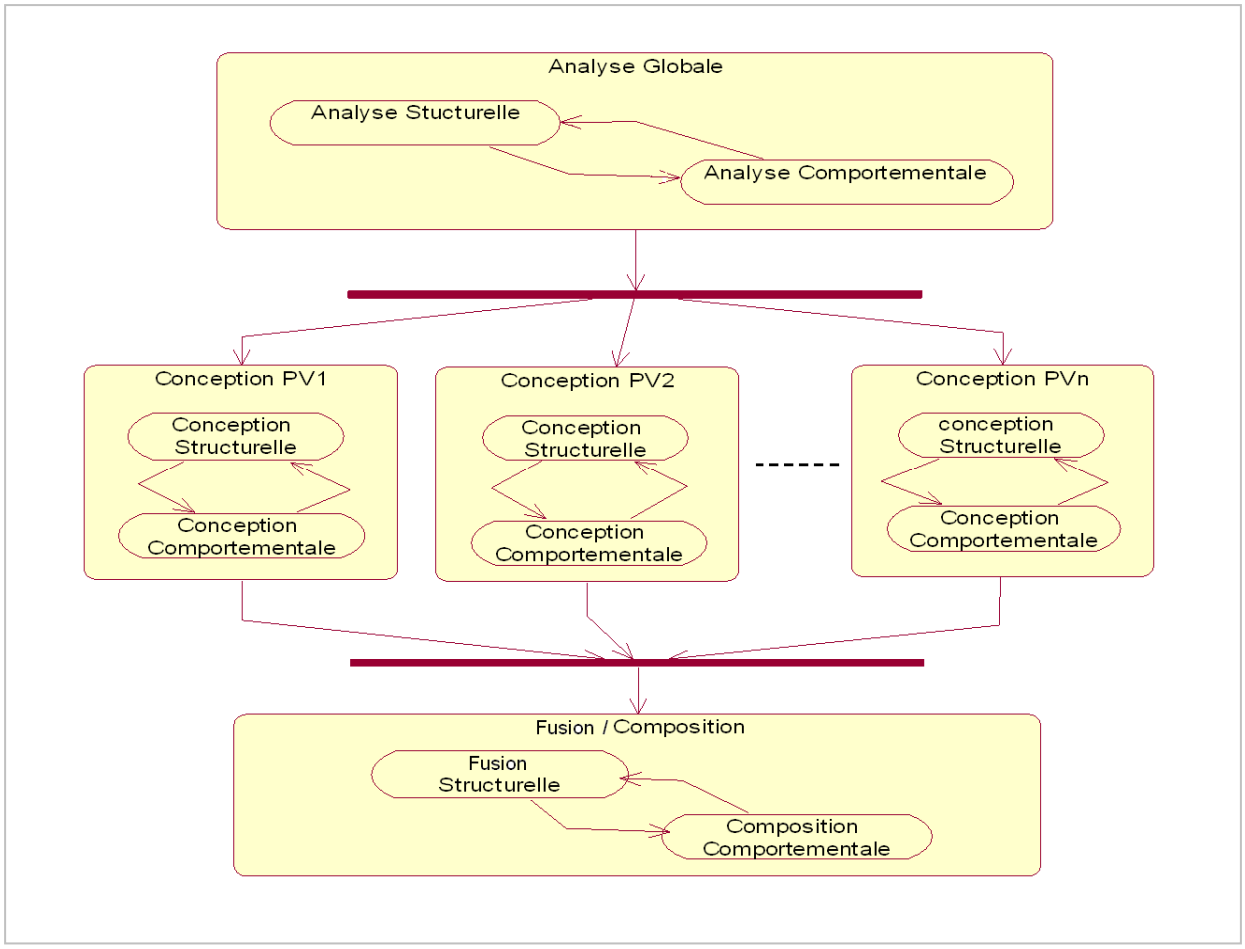


Figure V.1. Vision générale de la démarche VxUML

Nous détaillons dans les sous-sections suivantes les différentes phases de la démarche VxUML en explicitant les étapes constituant chaque phase ainsi que les résultats en sortie de chacune de ces phases.

V.2.1. Phase 1 : Analyse globale

La première phase de la démarche est l'analyse globale qui est une phase centralisée de modélisation des exigences. Son objectif principal est d'identifier les acteurs et d'extraire les besoins fonctionnels du système étudié, puis de les structurer sous forme de cas d'utilisations.

Le traitement structurel réside dans la réalisation des tâches suivantes :

– identification des acteurs (points de vue) ;
– extraction des fonctionnalités principales du système ;
– détermination des besoins des acteurs et des activités associées ;
– élaboration des diagrammes de cas d'utilisation en faisant apparaître les acteurs participants ;
– construction du glossaire de l'application.

Le traitement comportemental est constitué des tâches suivantes :

– représentation des responsabilités principales de chaque acteur en identifiant la participation de chacun dans les fonctionnalités principales du système. Ce traitement peut être accompli en se servant des diagrammes d'interactions et d'activités ;
– identification des principales collaborations entre acteurs en exploitant les diagrammes de collaborations.

Le résultat obtenu à la fin de cette phase est un premier modèle d'analyse de l'application, comportant notamment les cas d'utilisation pour les fonctionnalités de l'application, la liste des intervenants potentiels, leurs besoins, leurs collaborations et leurs participations dans les cas d'utilisation.

V.2.2. Phase 2 : Conception par points de vue

La deuxième phase est une conception décentralisée, au cours de laquelle plusieurs équipes de concepteurs peuvent travailler séparément pour réaliser des modèles par points de vue (appelés *modèles-vue*). Le résultat de cette phase est un ensemble de modèles UML, chacun répondant aux exigences d'un acteur à un niveau de granularité fin. Après l'établissement du modèle structurel selon un point de vue particulier, nous réalisons une analyse comportementale des classes réactives présentes dans ce modèle. Pour chacune de ces classes, nous définissons une machine à états capturant le cycle de vie de ses instances, les interactions avec l'environnement interne de l'application (les différents éléments du système) ainsi qu'avec l'environnement externe de l'application.

Pour chaque acteur (et donc point de vue) et en se basant sur le résultat de la première phase, le traitement structurel comporte les tâches suivantes :

– établissement des diagrammes de séquence réalisant les cas d'utilisation auxquels l'acteur considéré participe ;
– réalisation des diagrammes de classes préliminaires découlant des cas d'utilisation ;
– élaboration du diagramme de classes détaillé représentant les spécificités du point de vue courant ;

Une fois le diagramme de classes concernant un point de vue réalisé, nous démarrons une étape de spécification du comportement. Nous identifions les classes réactives du modèle-vue et nous représentons leurs comportements sous forme de machines à états que nous appelons *machines-vue*. Pour construire ces machines, nous procédons comme suit :

(a) identifier les états potentiels pour chaque classe réactive ;
(b) ajouter les transitions en adoptant le principe suivant :

1. si le comportement à spécifier ne peut pas être décrit uniquement à partir des éléments existant dans le modèle-vue mais dépend d'un ou d'autres points de vue, les conditions de déclenchement d'un comportement ne peuvent être précisées complètement. Pour remédier à ce problème, le développeur peut insérer des sondes sur le comportement recherché en déclarant des sondes abstraites dérivées de la classe racine *Probe* de la bibliothèque *ProbeLibrary* (cf. chapitre précédent). Le type concret de la sonde (une des sous-classes de la classe *Probe*, cf. Section IV.3.2.1) sera identifié plus tard dans la phase de fusion quand tous les éléments dont dépend la sonde seront connus. Une note détaillée – comportant les objectifs de la sonde – doit être associée à la déclaration de chaque sonde, pour être utilisée par le concepteur responsable de la phase de fusion.
2. si le comportement à spécifier peut être décrit en se basant sur les éléments existant dans le modèle-vue, nous utilisons les mécanismes habituels de communication inter-objets d'UML tels que l'échange de messages et de signaux. Néanmoins le concepteur peut définir des sondes concrètes (élémentaires ou composées) pour observer l'évolution de son modèle. La déclaration et l'utilisation de ces sondes ont été présentées dans la section IV.3.3.3.

Le résultat final de cette phase décentralisée est un ensemble de modèles dont chacun représente les spécificités d'un point de vue donné. Chaque modèle est représenté par des diagrammes structurels — principalement les diagrammes de classes —, ainsi que par des diagrammes comportementaux — principalement les machines-vue.

V.2.3. Phase 3 : Fusion

La troisième phase, centralisée, est la fusion des modèles-vue développés séparément. La fusion structurelle consiste à composer les modèles-vues afin d'aboutir au diagramme de classes VUML final. Le principe de cette fusion, qui peut être automatisée à l'aide de règles selon l'approche décrite dans [Anwar 08], est résumé comme suit :

- l'étape de mise en correspondance est guidée par un ensemble de règles dont le but est d'identifier les correspondances entre les éléments des modèles partiels. Les règles de correspondance sont basées sur les méta-propriétés définies dans les méta-classes du méta-modèle d'entrée UML [OMG, 2003a]. Les différentes relations de correspondance créées au cours de cette étape sont stockées dans un modèle séparé appelé *modèle de correspondances*.

- élaboration du modèle VUML global par composition des modèles initialement mis en correspondance en se basant sur un ensemble de règles de fusion et de translation. La stratégie de fusion s'appuie sur des règles prédéfinies détaillées dans [Anwar, 09]. L'idée est de

parcourir les liens de correspondance établis dans l'étape précédente et de déterminer selon le cas les règles de fusion à appliquer. Les éléments qui ne sont pas reliés sont simplement retranscrits dans le modèle VUML par le biais de règles de translation.

- enrichissement/optimisation manuelle du modèle VUML. Une fois le modèle VUML généré, on procède à son raffinage avec d'autres informations. En effet, durant cette opération, les éventuelles dépendances entres les vues doivent être explicitées afin de permettre par la suite d'assurer la cohérence du système. Ces dépendances sont modélisées dans VUML par des relations de dépendances stéréotypées par « viewDependency », et annotées par des contraintes exprimées en langage OCL.
- vérification de la conformité du modèle VUML produit par rapport à son méta-modèle. Cette étape consiste à vérifier les propriétés syntaxiques et sémantiques propres au profil VUML [Nassar, 2005]. Elle est nécessaire quand des changements manuels ont été effectués dans l'étape précédente, comme par exemple, l'ajout de relations de dépendance entre les classes vues.

La composition comportementale, effectuée après la fusion structurelle, consiste à composer le comportement des objets multivue (instances de classes multivue réactives) et à assurer la cohérence de l'ensemble. En fait, un objet multivue est composé d'un ensemble d'objets-vue (cf. section III.4.2.1) ayant chacun un comportement décrit par une machine-vue développée lors de la phase de conception par points de vue. La mise en cohérence et les liens entre les machines-vue se fait à travers la définition finalisée des observations qu'elles utilisent.

Le résultat final de cette phase de fusion est un modèle multivue VxUML répondant aux exigences de l'ensemble des acteurs du système. La partie structurelle de ce modèle est constituée principalement par le diagramme de classes VxUML composé de classes UML classiques et de classes multivue. La partie comportementale du modèle est basée sur les machines à états associées aux classes réactives. La spécification et la gestion de la cohérence de ces machines à états sont assurées par le mécanisme de sondes d'événements.

La section suivante explicite l'application de cette démarche à l'étude de cas "Gestion d'une agence de réparation de voitures".

V.3. Illustration de la démarche sur un cas d'étude

V.3.1. Présentation du cas d'étude

Nous avons choisi comme cadre applicatif de notre approche une application de gestion d'une agence spécialisée dans la réparation des voitures. Il s'agit d'un système d'information complexe, impliquant plusieurs acteurs et qui se prête à une conception par points de vue. Cette étude de cas détaille la construction d'un système selon la démarche VxUML décrite ci-dessus. Nous focalisons l'étude sur la spécification du comportement des objets multivue lors de la phase décentralisée et également sur la composition de ce comportement dans la phase de fusion. Nous limitons le périmètre de cette application aux sous-systèmes principaux suivants :

- Gestion des ressources matérielles : ce sous-système gère les biens de l'agence, ainsi que les stocks en termes de pièces de rechange utilisées dans les réparations de voitures.

- Gestion du personnel : ce sous-système gère les ressources humaines de l'agence, que nous restreindrons aux acteurs pertinents de l'agence, tels que les mécaniciens, électriciens, et responsables d'ateliers.
- Gestion financière : ce sous-système gère les tâches financières telles que les contrats avec les clients, les contrats avec les fournisseurs, et les dépenses internes.
- Gestion des voitures : ce sous-système traite la gestion des voitures admises dans les ateliers de l'agence pour réparation.

Le sous-système le plus important pour notre étude est celui de la gestion des voitures. C'est un sous-système nécessitant un traitement approfondi pour pouvoir spécifier son comportement. Les autres sous-systèmes sont plus liés à la gestion de données statiques sans présenter de caractéristiques comportementales importantes. De ce fait, ils seront traités d'une manière légère, mais suffisante pour fournir une étude complète suivant les étapes de la démarche VUML.

Le traitement du sous-système de gestion des voitures prend en compte la gestion des voitures, leurs états potentiels et leurs cycles de vie à l'intérieur de l'agence, les interactions entre les intervenants humains et les objets représentant les voitures, et plus généralement les différents échanges d'informations et de communications inter-objets. La classe Voiture constitue la cible de notre développement multivue dans le traitement des aspects structuraux (principalement à travers le diagramme de classes) et les aspects comportementaux (à travers les machines à états).

Compte tenu de la dimension importante de cette application, nous faisons des simplifications à chaque fois que cela est possible. Dans la suite de ce chapitre nous présentons en détail le développement des trois points de vue essentiels : *Client*, *ChefAgence* et *ResponsableAtelier*. Les détails donnés pour ces trois points de vue sont suffisants pour montrer comment se développent les modèles partiels par points de vue et comment se spécifient les interactions entre ces derniers pour définir le comportement du système.

V.3.2. Phase 1 : Analyse globale

L'objectif principal de cette phase est d'arriver à identifier les besoins des différents intervenants du système, et de les structurer en unités fonctionnelles sous forme de cas d'utilisations. Dans cette section, nous présentons les différentes étapes constituant cette phase, que ce soit au niveau structurel qu'au niveau comportemental. Nous illustrons toutes les étapes par des extraits de notre cas d'étude.

V.3.2.1. Identification des acteurs potentiels (points de vue)

Nous limitons les intervenants du système aux acteurs suivants :

- Le client propriétaire de la voiture
- Le chef d'agence
- Le mécanicien
- L'électricien
- Le responsable atelier

Pour des raisons de lisibilité et dans le but de simplifier le système étudié, nous ne prenons pas en considération certains intervenants tels que : le comptable de l'agence, la secrétaire d'accueil, etc.

V.3.2.2. Extraction des fonctionnalités principales du système

Nous résumons les fonctionnalités principales du système d'information de cette agence de réparation aux quatre fonctionnalités suivantes :

Gestion des voitures. Le système doit être capable de gérer les voitures à l'intérieur de l'agence en assurant les tâches suivantes :

- l'enregistrement des voitures et les clients,
- la gestion des opérations d'expertise et de réparation,
- l'enregistrement des pannes détectées,
- l'enregistrement des réparations effectuées,
- la gestion des tests des pannes réparées,
- la sauvegarde de l'historique de chaque voiture (historiques des pannes et des réparations effectuées).

Gestion des ressources matérielles. Le système doit gérer les ressources matérielles, comme :

- la gestion des affectations des véhicules de dépannage,
- la gestion des affectations des outils,
- la gestion des places dans les ateliers (pour réaliser les expertises, les réparations et les tests),
- la gestion des places du parking,
- la gestion des pièces de rechange : affecter les pièces, signaler l'épuisement du stock pour une pièce particulière, commander des pièces auprès du fournisseur, etc.

Gestion du personnel. Le système doit gérer les ressources humaines de l'agence, comme :

- l'affectation des mainteniciens aux voitures nécessitant une expertise,
- l'affectation des mainteniciens aux voitures nécessitant une réparation,
- l'affectation des mainteniciens aux voitures nécessitant un test.

Gestion financière. Les tâches de gestion financière incluent :

- la gestion des contrats avec les clients,
- la gestion des contrats avec les fournisseurs,
- la gestion des dépenses internes de l'agence.

V.3.2.3. Détermination des besoins des acteurs

Après l'identification des acteurs, on détermine leurs rôles au sein du système. Chaque acteur joue un rôle dans les fonctionnalités citées ci-dessus selon sa vision et attend des services adaptés à ses besoins. Le système doit répondre aux besoins spécifiques de chaque acteur par un ensemble de services que nous synthétisons comme suit :

- pour le client propriétaire de la voiture :
  - consulter les informations concernant sa voiture,
  - suivre son évolution dans la chaîne de réparation,
  - consulter les rapports d'expertise et de réparation,
  - négocier les contrats avec le responsable financier de l'agence,
  - effectuer le test de vérification avant la livraison de sa voiture.

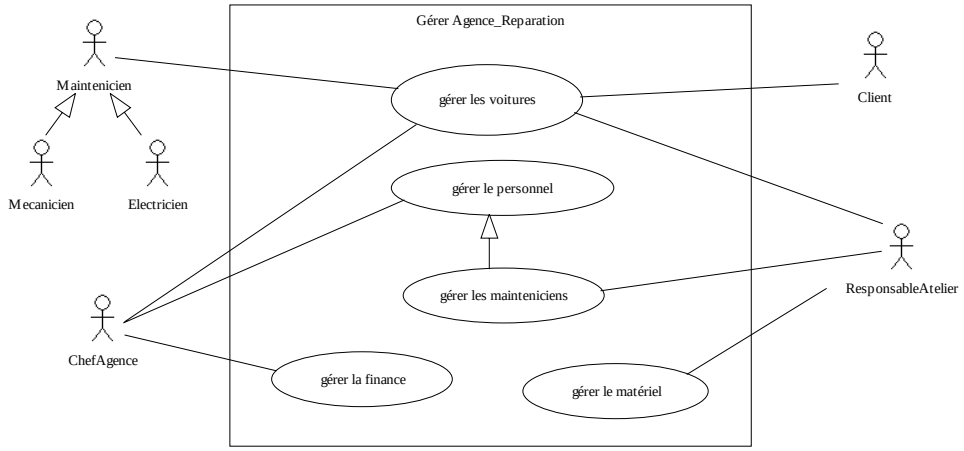
- pour le chef d'agence :
  - établir les contrats avec les clients,
  - établir les contrats avec les fournisseurs,
  - donner les ordres aux responsables pour démarrer l'expertise et la réparation,
  - faire le contrôle financier.

- pour le mécanicien/électricien :
  - consulter l'historique des pannes,
  - rédiger et enregistrer les rapports d'expertise et de réparation,
  - pour chaque panne, enregistrer les pièces changées,
  - enregistrer les détails des réparations apportées.

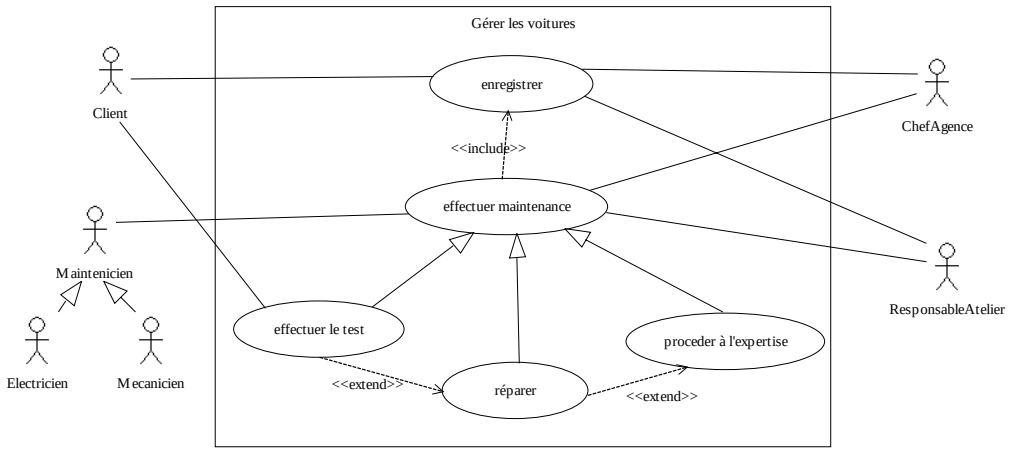
- pour le responsable atelier :
  - affecter les mainteniciens (électriciens/mécaniciens),
  - affecter les ressources matérielles (véhicule de dépannage, outils de travail),
  - affecter les pistes pour effectuer les expertises, les réparations et les tests,
  - gérer le parking,
  - gérer et affecter les pièces de rechange.

V.3.2.4. Elaboration des cas d'utilisation

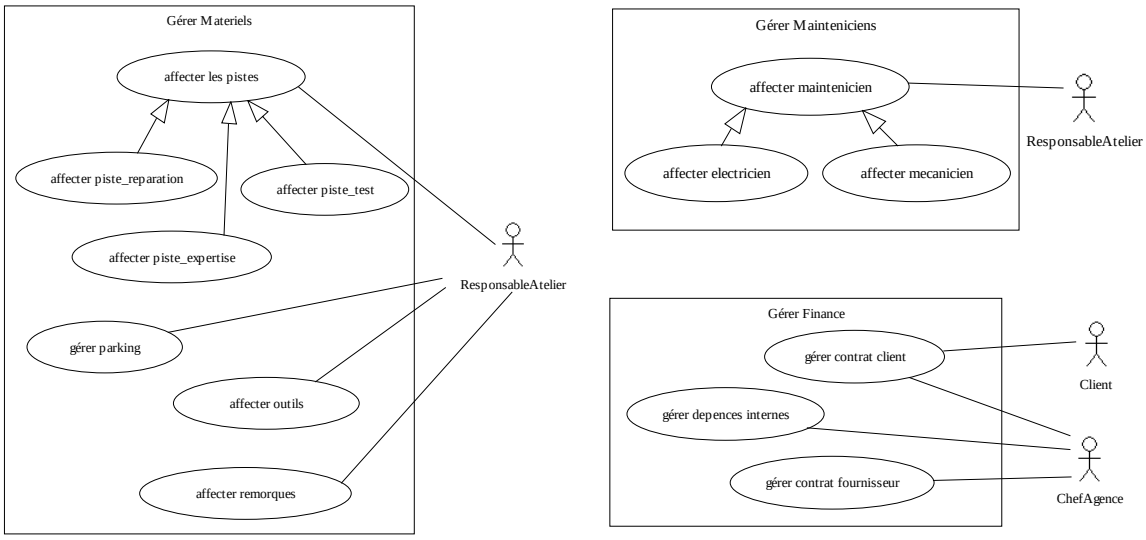
La Figure V.2-a donne un aperçu général des fonctionnalités que le système doit assurer avec les acteurs responsables. Ces fonctionnalités sont : (1) la gestion des voitures, (2) la gestion des mainteniciens, (3) la gestion des ressources matérielles, et (4) la gestion financière.



(a) - Cas d'utilisation racine de l'application "Gérer Agence_Réparation"



(b) - Cas d'utilisation "Gérer les voitures"



(c) - Cas d'utilisation "Gérer Matériels", "Gérer Mainteniciens" et "Gérer Finance"

Figure V.2. Les principaux cas d'utilisation principaux de l'application

La Figure V.2-b et la Figure V.2-c détaillent les cas d'utilisation mentionnés dans la Figure V.2-a avec les acteurs responsables. Pour le cas d'utilisation "gestion des voitures", gérer les voitures de l'agence revient à gérer leur enregistrement, les expertises et les réparations apportées à chacune d'elles, et les tests de vérification finals. Le client intervient dans le cas d'utilisation "enregistrer" par la communication des informations concernant sa voiture, il participe avec le chef d'agence à la réalisation des contrats d'expertise et de réparation et valide la réparation en effectuant le test final du bon fonctionnement de sa voiture. Les mainteniciens interviennent dans les activités techniques telles que l'expertise, la réparation et les tests. La Figure V.2-c présente les cas d'utilisation "Gérer matériel", "Gérer personnel" et "Gérer finance".

Nous rappelons le lecteur que dans cette approche, nous n'identifions pas les classes multivue dans cette phase ni leurs machines à états globales comme ce que nous faisions dans la première approche (cf. section0 IV.2). En fait, nous avons critiqué l'identification précoce de ces éléments dans la section IV.2.3. Dans l'approche adoptée à base de sonde d'événements ces contraintes sont dépassées et donc le développement des vues dans la phase décentralisée se fait d'une manière totalement indépendante les unes aux autres.

V.3.3. Phase 2 : Analyse/Conception par points de vue

La phase de conception structurelle par points de vue consiste à faire des conceptions partielles et décentralisées du système étudié, chacune développant les besoins d'un acteur. En effet, à un acteur correspond un point de vue et une conception partielle modélise le système selon le point de vue d'un acteur donné. Les conceptions partielles se basent sur les résultats de l'analyse globale. Chacune commence par la reprise des diagrammes développés lors de la première phase – principalement les cas d'utilisation – dans lesquels intervient l'acteur associé au point de vue traité. Sur cette base nous établissons les diagrammes de séquence et d'activités qui détaillent les scénarios de réalisation de ces cas d'utilisation. Ensuite, nous établissons des diagrammes de classes préliminaires, de façon itérative, et les enrichissons jusqu'à l'obtention du diagramme de classes détaillé représentant les spécificités de l'acteur considéré.

Le résultat de cette seconde phase est un ensemble de modèles UML complets (traitant à la fois les aspects structurel et comportemental), chacun de ces modèles représentant les spécificités d'un acteur.

Dans les sections qui suivent, nous illustrons les étapes de la démarche VxUML pour le point de vue *Client*. Nous donnons à la fin de la section le résultat de la modélisation des autres points de vue en appliquant les mêmes étapes que celles suivies dans cette phase.

V.3.3.1. Établissement des diagrammes de séquence réalisant les cas d'utilisation – acteur *Client*

En se basant sur les exigences identifiées lors de la première phase, nous détaillons les fonctionnalités principales relatives au point de vue *Client*. Pour simplifier l'étude, nous limitons le développement à deux scénarios représentés par des diagrammes de séquence : (1) Enregistrement d'une voiture, et (2) Réparation d'une voiture.

V.3.3.1.1. Diagramme de séquence "Enregistrement d'une voiture" du point de vue Client

La procédure suivie par un nouveau client lors de la prise en charge de sa voiture par l'agence se résume aux trois étapes suivantes : le client (1) établit un premier contact avec l'agence de réparation, (2) communique ses informations personnelles, telles que son nom, son téléphone, etc., (3) communique les informations concernant sa voiture en panne, notamment son numéro d'immatriculation et l'adresse où elle est en panne.

Après l'accomplissement de ces étapes, la voiture sera enregistrée dans le système d'information de l'agence. Ces étapes sont illustrées dans le diagramme de séquence de la Figure V.3. A partir de ce diagramme de séquence, on peut tirer un ensemble d'informations qui vont servir à l'enrichissement du diagramme de classes du point de vue *Client*, par exemple :

- pour la classe *Voiture*, on déduit les attributs : numéro d'immatriculation *mat* et son adresse où est tombée en panne *adrEnPanne*,
- pour la classe *AgenceReparation*, on déduit la méthode publique *contacterAgence()*, les méthodes privées *enregistrerClient()* et *enregistrerVoiture()* et aussi les attributs publics *adresse* et *tel*.
- pour la classe *Client*, on déduit les attributs : *nom*, *adresse* et *tel*.

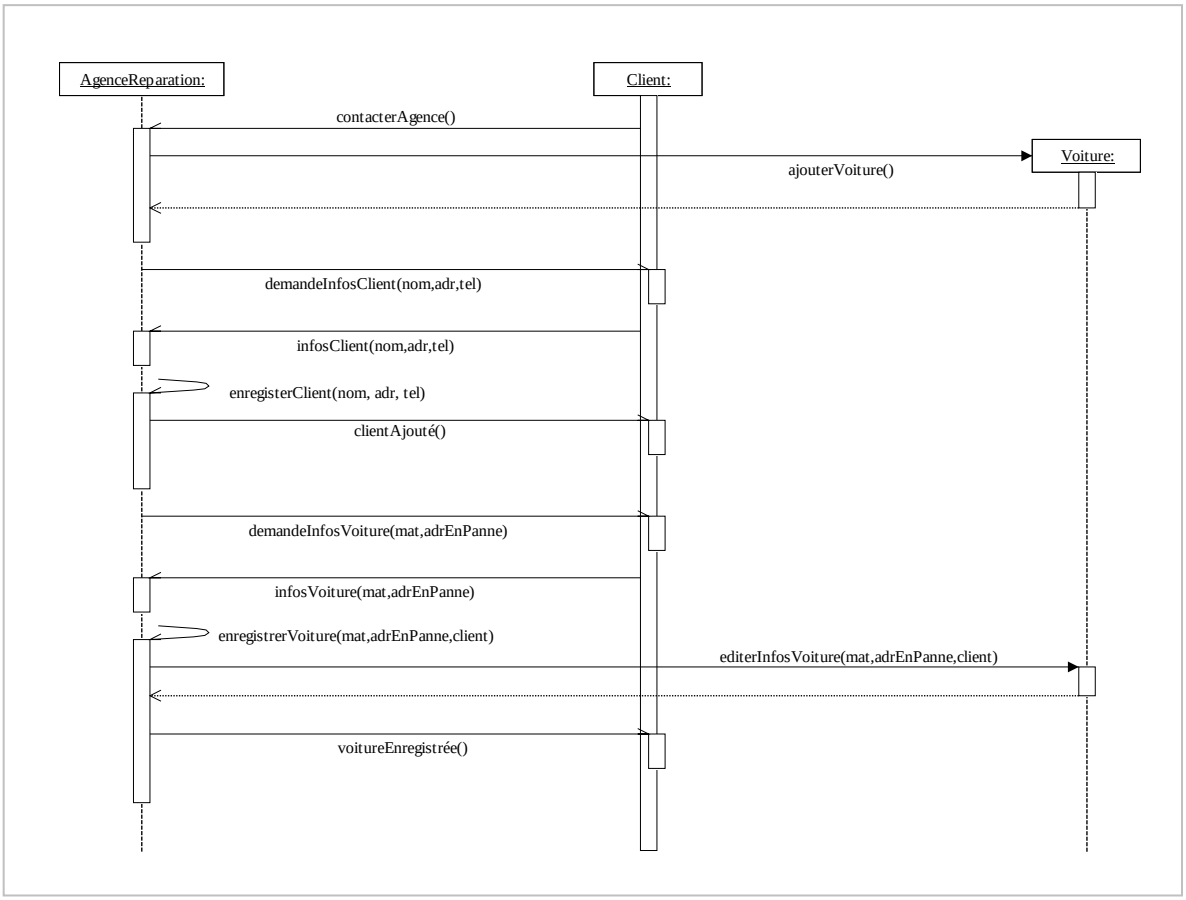


Figure V.3. Diagramme de séquence "Enregistrement d'une voiture"

V.3.3.1.2. Diagramme de séquence "Réparation d'une voiture" du point de vue Client

La Figure V.4 présente le diagramme de séquence exprimant le scénario suivi pour établir la réparation d'une voiture selon le point de vue *Client*. Ce scénario se résume comme suit :

- le client récupère le rapport de l'expertise,
- il consulte les résultats,
- il traite les options de réparation possibles,
- si plusieurs choix sont proposés, le client choisi l'option de réparation convenable,
- il communique ses choix et attendre le précontrat,
- il réceptionne le précontrat,
- par rapport aux termes du contrat et notamment les frais de réparation, le client peut :
    - accepter ces conditions : la négociation du contrat a réussi et la réparation de la voiture peut commencer ;
    - refuser ces conditions : la négociation du contrat a échoué, et la voiture doit quitter l'agence.

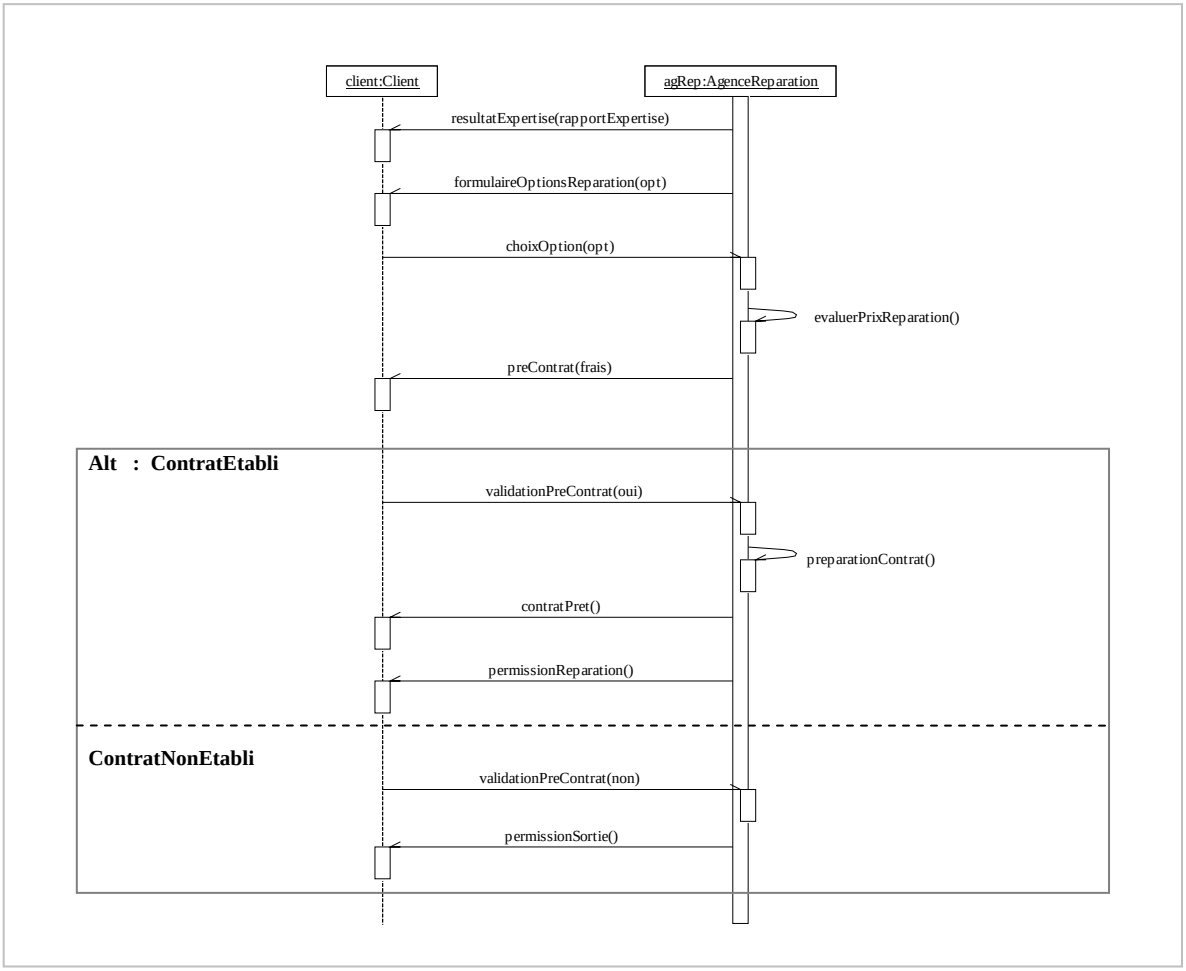


Figure V.4. Diagramme de séquence "Réparation d'une voiture"

A partir de ce diagramme de séquence, on peut tirer des informations qui vont enrichir le diagramme de classes associé au point de vue *Client*. Par exemple, pour la classe Voiture, on identifie la méthode *afficherResultatExpertise()* et les attributs *ficheReparation*, *dateSortie* et *contratEtabli*.

V.3.3.2. Développement du diagramme de classes détaillé – acteur client

En nous basant sur le cahier des charges de l'application, sur les résultats de la phase d'analyse globale et également sur les diagrammes développés précédemment pour le point de vue *Client*, nous avons produit le diagramme de classes représenté dans la Figure V.5 ci-dessous. Les classes importantes dans ce diagramme sont : *Voiture*, *AgenceReparation*, *ContratReparation* et *OperationEntretien*. Nous signalons que pour des raisons de lisibilité du diagramme nous ne présentons que les classes indispensables à la réalisation des besoins associés à l'acteur client et notamment celles nécessaires pour l'expression des besoins comportementaux. Cette simplification est faite également au niveau de la définition des classes ; seule la classe Voiture est définie d'une manière détaillée. L'objectif principal du système, selon la vision du client, est de gérer l'évolution des objets de type Voiture durant leurs cycles de vie, commençant par l'admission au garage, passant par les différentes étapes de la chaîne de réparation, jusqu'à la sortie finale du garage.

La classe *AgenceReparation* est considérée comme la classe principale du système ; elle crée la coordination entre les différents sous-systèmes de l'application qui sont : (i) le sous-système de gestion des ressources matérielles, (ii) le sous-système de gestion du personnel, (iii) le sous-système de gestion financière.

Les classes *Expertise*, *Reparation* et *Test* sont des classes associées à la classe *Voiture*. Elles vont servir à la représentation et à la sauvegarde des détails des opérations de maintenance effectuées sur la voiture.

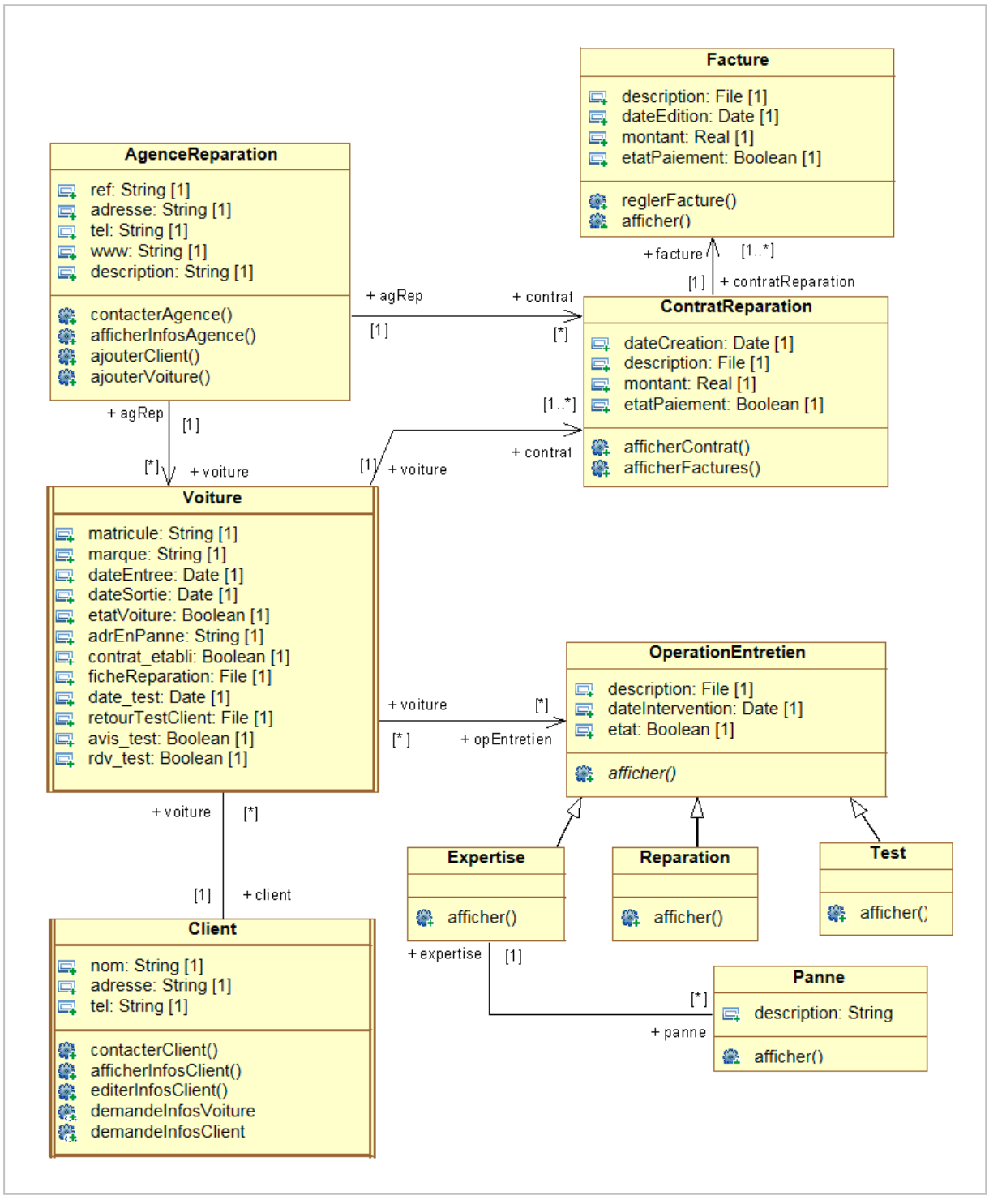


Figure V.5. Diagramme de classes UML du point de vue Client

V.3.3.3. Spécification comportementale du modèle-vue *Client*

Une fois le diagramme de classes concernant un point de vue réalisé, nous démarrons une étape de spécification du comportement. Cette étape de spécification comportementale commence par l'identification des classes réactives du modèle-vue. D'après l'analyse des cas d'utilisation et des besoins de l'acteur client, seule la classe Voiture possède un comportement réactif, par conséquent, son comportement doit être spécifié par une machine à états. Les autres classes du modèle sont considérées comme des classes statiques de données.

Suite à l'étape d'enregistrement de la voiture, le système crée automatiquement un objet de type *Voiture*. Cet objet gère le cycle de vie de la voiture depuis son enregistrement dans le système jusqu'à sa sortie du garage passant par la chaîne de réparation. Il réalise une double communication avec l'environnement : *interne* avec le système et *externe* avec les acteurs. L'objet *Voiture* est détruit par le système une fois la voiture sortie de l'agence. Avant sa destruction, le système doit sauvegarder l'historique de cette voiture avec la liste des pannes détectées et la liste des réparations effectuées. En suivant les étapes de la démarche et en nous basant sur l'ensemble des informations disponibles dans le modèle du point de vue *Client*, nous aboutissons à la machine à états représentée dans la Figure V.6. Nous avons omis le développement de certains états tels que *Negociation_contrat_reparation*, *En_test* car ils nécessitent plus de raffinement et de détails, ce qui sort de l'objectif principal de ce chapitre.

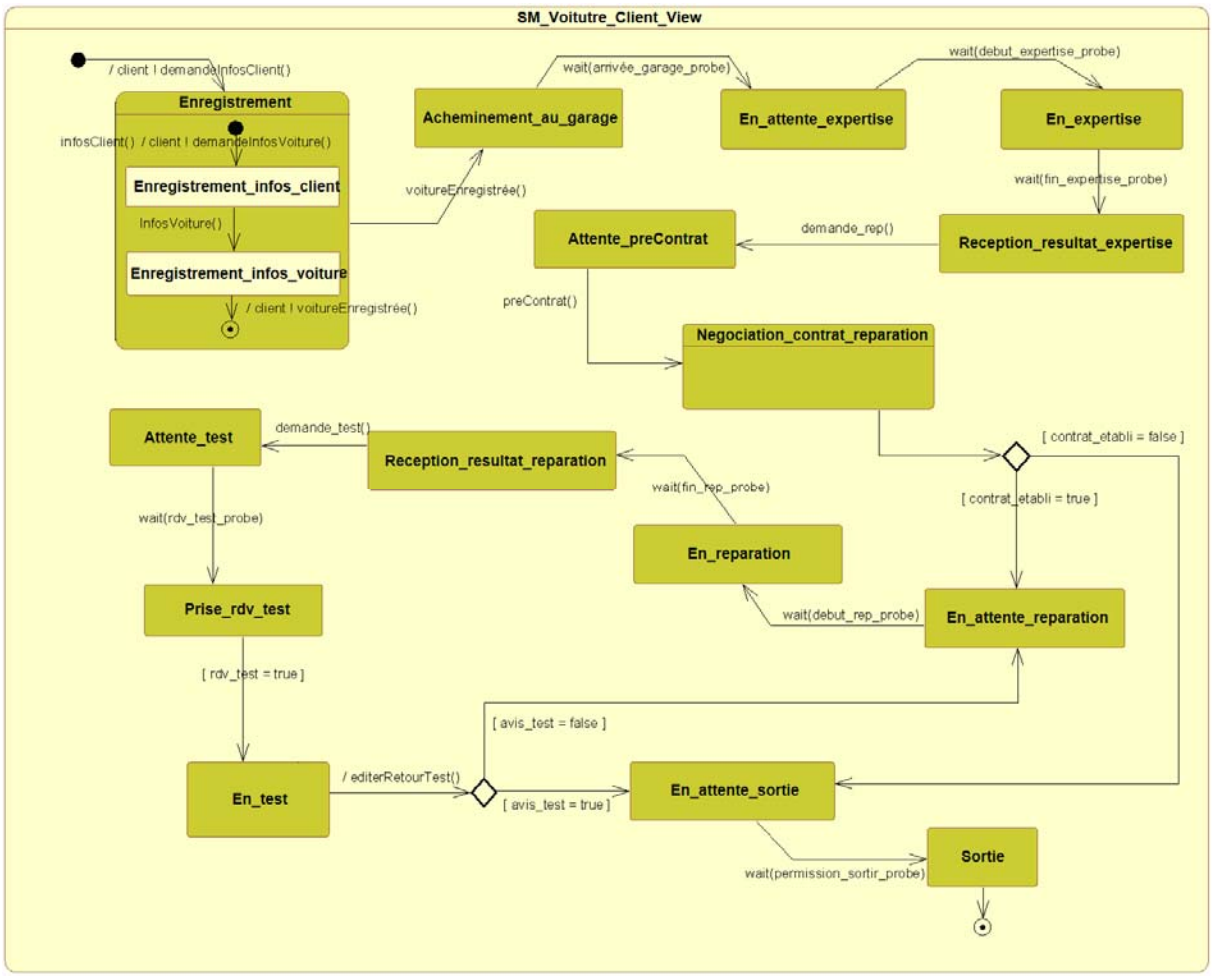


Figure V.6. Machine à états de la classe Voiture pour le point de vue Client

Dans cette étape de spécification du comportement sous forme de machines à états, nous avons utilisé les moyens offerts par UML, notamment la communication à travers les signaux tels que *demande_reparation* et *infosClient*. En conséquence il faut déclarer les signaux utilisés par la machine à états (partie supérieure de la Figure V.7) ainsi que les réceptions dans la classe associée (cf. Figure V.8). Nous avons regroupé les signaux utilisés dans le point de vue *Client* dans le paquetage *Signal_Package_Client_Viewpoint*, et nous avons regroupé l'ensemble des sondes d'événements dans le paquetage *Probe_Package_Client_Viewpoint* (cf. Figure V.7).

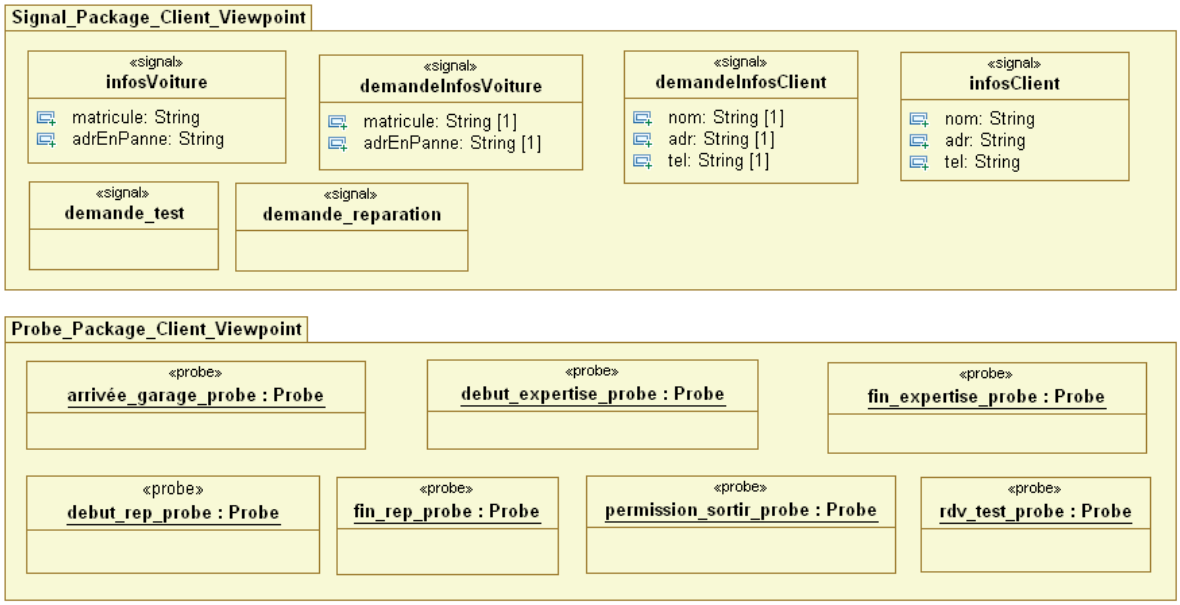


Figure V.7. Paquetages des signaux et des sondes du modèle - point de vue Client

Dans le cas où le comportement dépend des autres points de vue, nous avons déclaré les sondes nécessaires pour référencer les informations utiles. Par exemple, la transition depuis l'état *En_attente_reparation* vers l'état *En_reparation* est déclenchée par l'événement de démarrage réel de la réparation. Or cette information n'est pas disponible dans le modèle-vue *Client* car on ne connaît pas dans ce modèle l'acteur responsable de l'action associée au démarrage de la réparation ; on ne connaît pas non plus la manière dont cette action va être représentée. Pour ce faire, nous avons déclaré la sonde *debut_rep_probe* (cf. partie inférieure de la Figure V.7). La Figure V.8 montre que la classe Voiture référence un ensemble de sondes par le biais du stéréotype « *probeUse* ».

La plupart des déclarations de sondes dans cette phase de la démarche VxUML sont abstraites du fait que les paramètres des comportements recherchés sont indéfinis dans cette phase et ne seront identifiés que lors de la phase ultérieure de fusion. Toutefois, la liste de sondes est complète et délimite l'ensemble des interactions possibles avec les autres modèles-vue après la fusion. En principe, un modèle-vue de ce type est déjà un modèle complet qui peut, avec un outillage adapté permettant de simuler l'activation des sondes, être exécuté dans le but d'être validé. Nous mettons ce point dans les perspectives de ce travail de thèse.

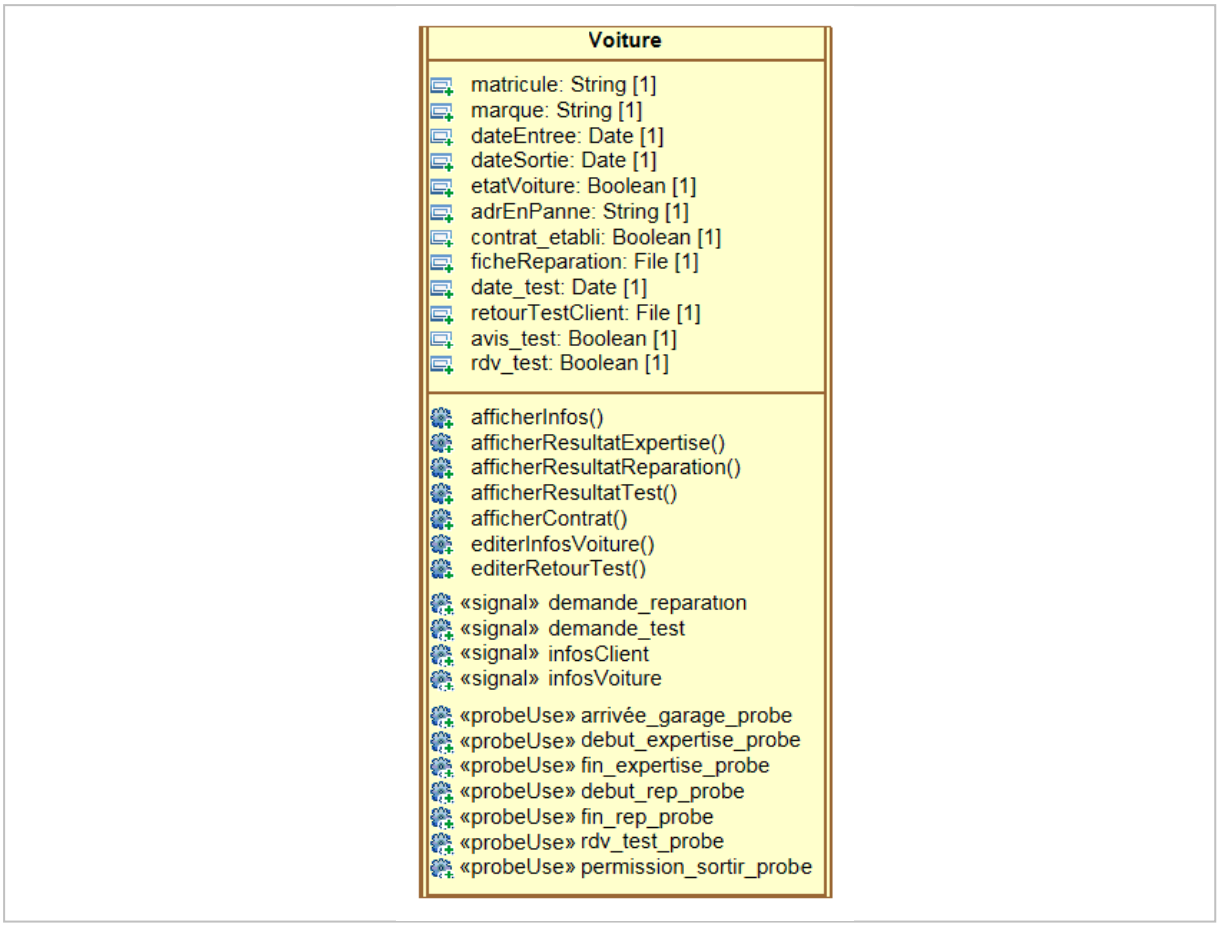


Figure V.8. Définition complète de la classe Voiture du point de vue Client

V.3.3.4. Synthèse des différents modèles résultant de la phase de conception par points de vue

En suivant la même démarche que celle utilisée ci-dessus pour le point de vue *Client*, nous avons élaboré les modèles structurels et comportementaux des autres points de vue. Les trois sous-sections suivantes donnent une synthèse des résultats du développement des modèles structurels et comportementaux des points de vue *ChefAgence* et du *ResponsableAtelier*.

V.3.3.4.1. Point de vue ChefAgence

Diagramme de classes du point de vue *ChefAgence*

Dans le diagramme de classes associé au point de vue *ChefAgence* (cf. Figure V.9), nous remarquons que les fragments liés à la supervision de l'agence et à la gestion financière sont beaucoup plus développés par rapport au point de vue *Client*. En effet, ces fragments expriment des fonctionnalités représentant des préoccupations importantes pour le chef d'agence. Néanmoins nous avons réduit le modèle aux éléments pertinents en liaison avec la tâche de gestion des voitures notamment les classes Voiture, AgenceReparation et DossierFinancier.

Nous avons représenté les besoins et les fonctions (en liaison avec la gestion des voitures) du chef d'agence au niveau de son diagramme de classes de la façon suivante :

- les ordres d'acheminement de la voiture en panne vers le garage, l'ordre d'expertise, l'ordre de réparation, l'ordre de test et l'ordre de livraison sont représentés dans la classe Voiture par les attributs suivants : permissionAchemin, permissionExp, permissionRep, permissionTest et permissionSortir ;
- la gestion financière est assurée par la classe DossierFinancier et ses trois sous-classes. La classe ContratFournisseur présente les contrats avec les fournisseurs de l'agence, lors que la classe DepenseInterne représente les dépenses nécessaires au fonctionnement de l'agence. La gestion des contrats de réparation avec les clients est représentée par la classe ContratReparationClient et également à travers l'attribut contrat_etabli de la classe Voiture. On associe à chaque type de DossiersFinancier un ensemble de factures à travers la classe Facture ;
- on organise le test de validation du client à travers les attributs suivants de la classe Voiture : rdv_test et test_valide.

La classe OperationMaintenance ainsi que ses sous-classes OperationExpertise, OperationReparation et OperationTest sont également importantes pour le point de vue *ChefAgence* dans la mesure où elles donnent une vision sur les opérations effectuées sur la voiture, celles qui sont en cours et celles qui sont planifiées ; elles permettent également d'estimer le coût des opérations à travers l'opération estimerCoutOperation de la classe OperationMaintenance.

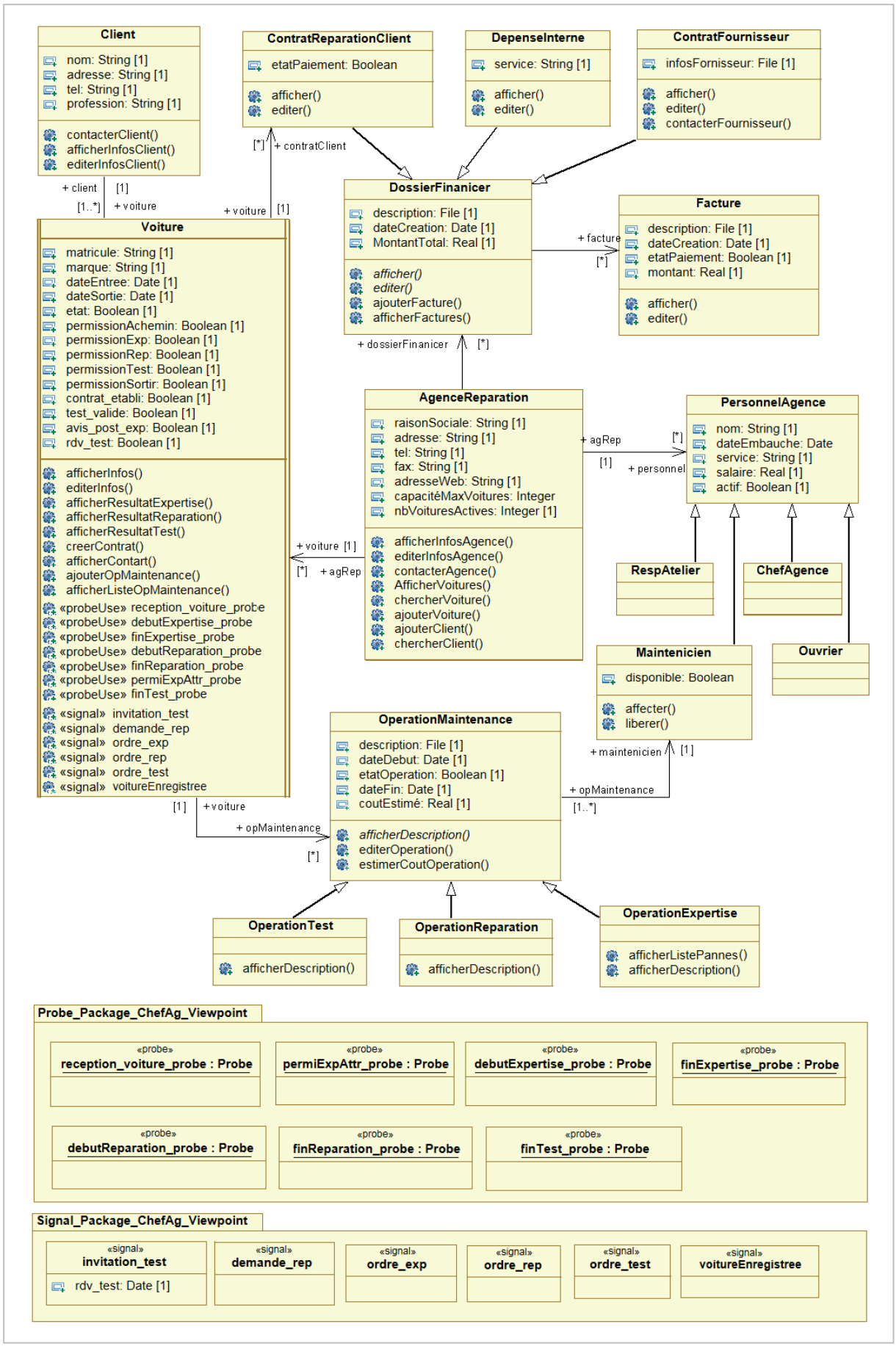


Figure V.9. Diagramme de classes UML – point de vue ChefAgence

Machine-vue de la classe Voiture pour le point de vue *ChefAgence*

La machine à états représentée dans la Figure V.10 exprime le comportement des objets de type Voiture selon la vision du chef d'agence. Au cours du développement de cette dernière, plusieurs moyens de spécification de comportement ont été utilisés.

Communication par utilisation d'attributs. Le fait de rendre certains attributs publics dans une classe permet aux objets du système d'accéder directement à ces attributs. Nous avons utilisé ce mode de communication au sein de la machine à états de la classe Voiture à plusieurs reprises comme par exemple les attributs booléens permissionAchemin et permissionSortir. Ces deux attributs expriment, respectivement, que le chef d'agence a donné sa permission pour l'acheminement de la voiture en panne, et pour la livraison de la voiture à son propriétaire.

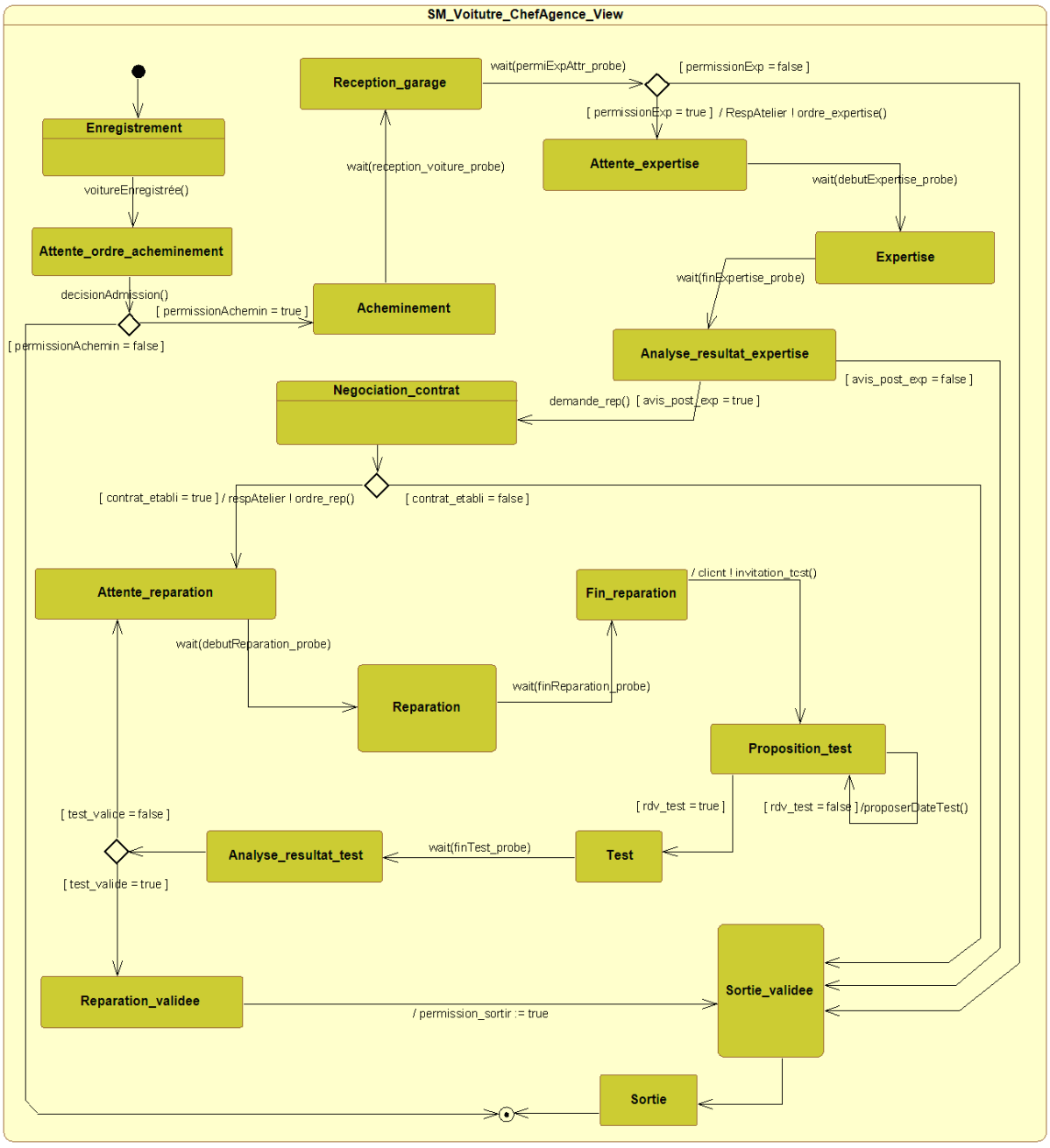


Figure V.10. Machine à états de la classe Voiture pour le point de vue ChefAgence

Communication par échange de signaux. Nous avons utilisé également ce mode de communication basé sur l'échange de signaux. Par exemple, c'est le cas du signal voitureEnregistree envoyé par la classe AgenceReparation suite à l'enregistrement de la voiture et utilisé par la machine à états pour démarrer le processus d'acheminement de la voiture en panne. La Figure V.9 donne l'ensemble des signaux utilisés dans le modèle comportemental du point de vue *ChefAgence* ; nous avons regroupé ces signaux dans le paquetage Signal_Package_*ChefAgence*_Viewpoint.

Communication par utilisation de sondes d'événement. Nous avons déclaré un ensemble de sondes abstraites dans les cas où le comportement n'a pas pu être spécifié. Nous donnons deux exemples de sondes utilisées dans la machine à états développée :

- la transition depuis l'état "Acheminement" vers l'état "Reception_garage" est déclenchée par l'événement de l'arrivée de la voiture au garage après son acheminement. Cependant, cette activité ne rentre pas dans les responsabilités du chef de l'agence et aucune information permettant d'identifier cet événement n'est disponible dans les modèles associés au chef de l'agence. Par conséquent, pour pouvoir continuer la spécification comportementale et pallier ce manque d'information, nous avons déclaré la sonde *reception_voiture_probe*.
- La sonde *fin_test_probe* est déclarée pour informer le chef d'agence du moment de la réalisation du test de validation des réparations effectuées sur la voiture.

Le référencement de ces sondes par la classe Voiture est montré dans la Figure V.9 par le stéréotype « *probeUse* ». Les sondes utilisées dans le modèle comportemental du chef d'agence sont regroupées dans le paquetage *Probe_Package_ChefAgence_Viewpoint*.

V.3.3.4.2. Point de vue ResponsableAtelier

Diagramme de classes du point de vue *ResponsableAtelier*

Le diagramme de classes associé au point de vue *ResponsableAtelier* est représenté par la Figure V.11. Le responsable d'atelier s'intéresse aux fonctionnalités liées à la gestion du matériel, du personnel de l'agence et le suivi du travail des mainteniciens. Dans ce diagramme, nous faisons également les simplifications faites au niveau des autres points de vue, et nous ne présentons que les classes pertinentes en liaison avec la tâche de gestion des voitures. Les classes clés de ce diagramme sont : Voiture, AgenceReparation, OperationMaintenance, Personnel et RessourceAgence.

La tâche de gestion du matériel est représentée par la classe RessourceAgence et également par ses classes dérivées telles que Parking, Atelier, Outil et PieceRechange. Ces classes sont munies des opérations *affecter()* et *liberer()* et de l'attribut booléen *disponibilité* qui permettent de gérer la réservation des différentes ressources pour l'utilisation dans les opérations de maintenance.

La gestion du personnel est assurée par la classe Personnel. Le responsable d'atelier a la tâche d'assurer l'affectation des mainteniciens (mécanicien ou électricien) aux voitures en cours d'une opération de maintenance. Il a également la tâche de suivi du travail de ces mainteniciens en leur donnant les ordres concernant le début de l'opération de maintenance, la rédaction des rapports de ces opérations, la gestion du stock de pièces de rechange, etc.

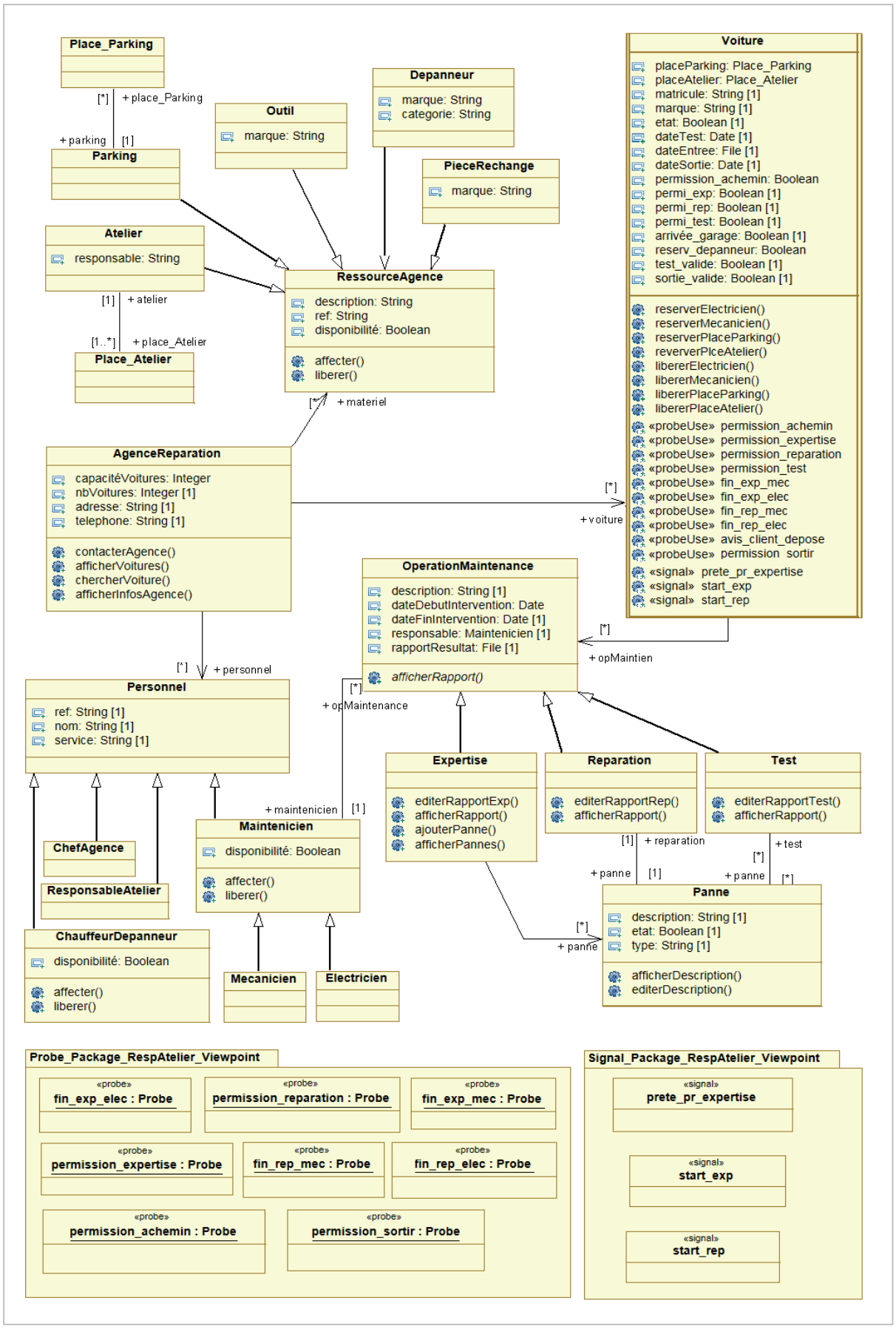


Figure V.11. Diagramme de classes UML : point de vue ResponsableAtelier

Machine-vue de l'objet Voiture pour le point de vue *ResponsableAtelier*

Nous représentons dans la machine à états de la Figure V.12 les états/transitions essentiels à la réalisation du comportement des objets de type Voiture selon la vision d'un responsable d'atelier. Cette vision a un objectif principal qui est la préparation du contexte nécessaire aux éventuelles opérations de maintenance à effectuer sur la voiture.

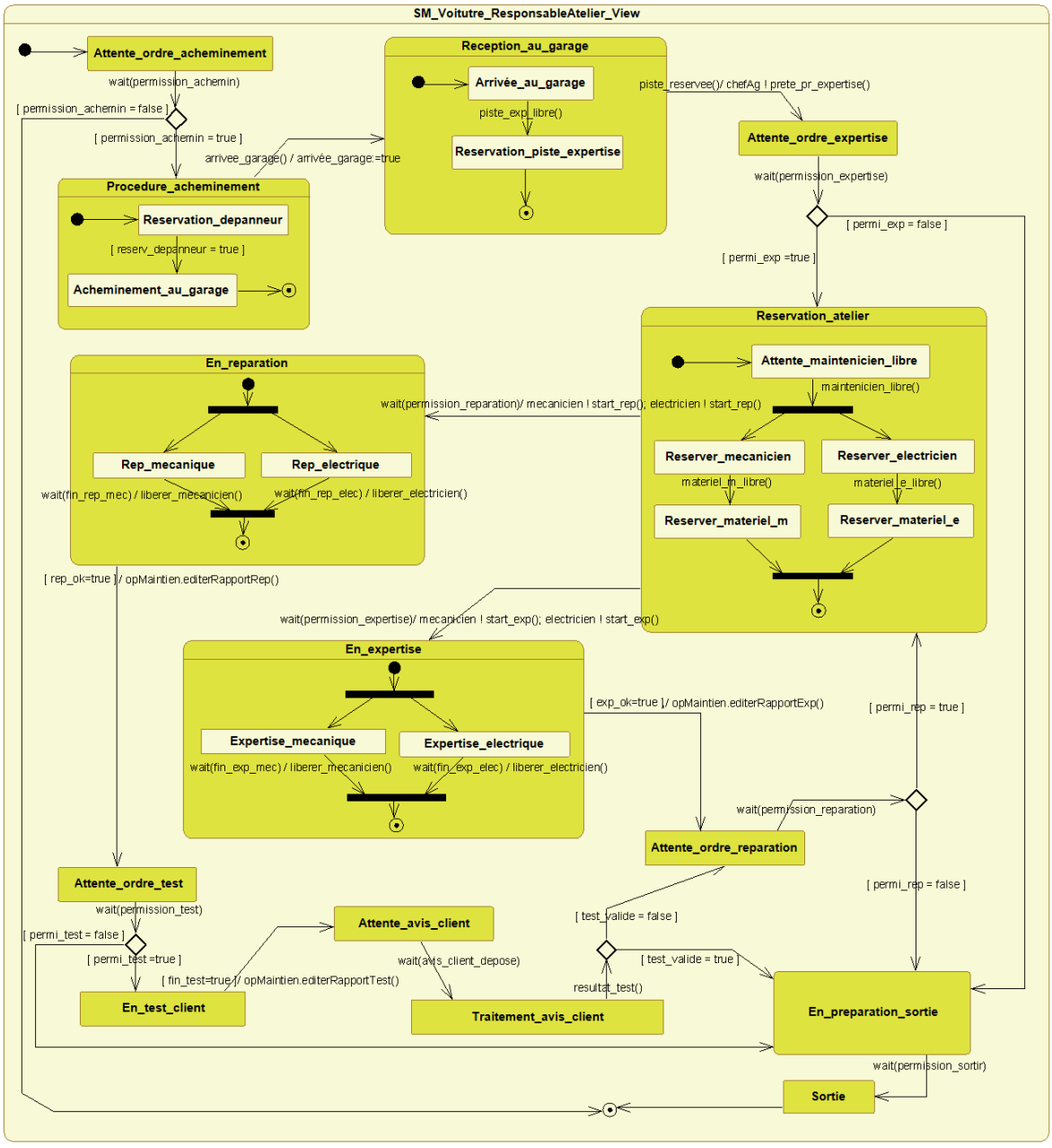


Figure V.12. Machine à états de la classe Voiture pour le point de vue ResponsableAtelier

Le cycle de vie commence par l'attente de la permission d'acheminement de la voiture vers le garage ; cela est réalisé par la déclaration de la sonde *permission_achemin_probe* qui va permettre de mettre à jour l'attribut booléen *permission_achemin*. Si la réponse est favorable, la voiture entre dans l'état *Procedure_acheminement*. Après sa réception dans le garage et après réservation d'une place dans l'atelier pour effectuer l'expertise, on attend la permission du chef d'agence, information qui va être observée par la sonde *permission_expertise_probe*. Suite à cette décision, le responsable d'atelier effectue

les réservations nécessaires au bon déroulement de l'opération d'expertise telles que la réservation des mainteniciens et des outils adéquats, et donne ensuite les ordres à ces mainteniciens de commencer leur travail. Une fois l'expertise réalisée, le responsable d'atelier participe à l'édition du rapport de l'opération et la voiture entre dans un état *Attente_ordre_reparation*. Nous avons déclaré la sonde *permission_rep_probe* pour observer l'événement traduisant cette permission. Après la réponse favorable de réparation, on effectue à nouveau les réservations nécessaires. Quand la réparation est faite, le cycle de vie se termine par le test de vérification des réparations effectuées pour attendre la permission de livraison de la voiture à son propriétaire à travers la sonde *permission_sortir_probe*.

La Figure V.11 donne un aperçu des signaux utilisés dans le modèle comportemental du point de vue *ResponsableAtelier*, regroupés dans le paquetage *Signal_Package_ChefAgence_Viewpoint*. Le référencement des sondes utilisées par la classe Voiture est montré dans la Figure V.11 par le stéréotype « *probeUse* ». L'ensemble des sondes utilisées dans le modèle comportemental du point de vue *ResponsableAtelier* sont regroupées dans le paquetage *Probe_Package_ChefAgence_Viewpoint*.

V.3.4. Phase 3 : Fusion/composition des modèles-vue

Dans cette section, nous explicitons la démarche VxUML concernant la troisième phase de fusion. La fusion structurelle consiste à produire le modèle global VxUML qui sera partagé par l'ensemble des acteurs du système. Ce modèle global est constitué principalement par le diagramme de classes VxUML composé de classes UML classiques et de classes multivue. Par contre, pour les modèles de comportement, il s'agit d'une composition plutôt que d'une fusion, car on ne converge pas vers un modèle unique mais on garde les modèles de comportement par points de vue. Ces modèles de comportement sont basés principalement sur les machines à états associées aux classes réactives. La spécification et la gestion de la cohérence de ces machines à états sont assurées par le mécanisme de sondes d'événements.

V.3.4.1. Fusion structurelle

Le processus suivi dans cette phase de fusion structurelle a été décrit dans la thèse de A. Anwar [Anwar 09]. Le concept de fusion peut être défini comme une opération de transformation de deux éléments ou plus en entrée pour donner un élément en sortie sans qu'il y ait de redondance d'information dans l'élément en sortie. Nous illustrons dans cette section le principe de cette fusion (cf. V.2.3).

La première étape de la fusion est la mise en correspondance des modèles d'entrée (diagrammes de classes par points de vue) qui peut révéler des ambiguïtés ou des incohérences sur les classes (nommage, attributs, méthodes). Ces incohérences sont traitées manuellement ou de façon semi-automatique par le concepteur responsable de la fusion. Dans une seconde étape, les diagrammes de classes par points de vue sont traités pour donner lieu à un seul diagramme de classes VUML sans perdre les spécificités de chaque acteur.

Le résultat final de la fusion structurelle est représenté par le diagramme VUML de la Figure V.14. Les classes multivues de cette application sont : Voiture, AgenceReparation, OperationMaintenance, Panne, ContratClient, et Personnel. Chacune de ces classes offre des vues spécifiques adaptées aux besoins des utilisateurs du système. Notons que, pour des soucis de lisibilité,

nous n'avons détaillé que la classe multivue Voiture (cf. Figure V.13) et que les autres classes multivue sont représentées d'une manière classique sans faire apparaître leurs vues. Les vues associées à la classe Voiture sont celles des acteurs chef d'agence, client et responsable atelier. La partie partagée d'une instance de Voiture – stéréotype « base » – regroupe les attributs, les méthodes, les signaux et les sondes accessibles par tous les acteurs. La partie spécifique à chaque acteur – stéréotype « view » – représente les attributs, les méthodes, les signaux et les sondes qui ne sont accessibles que par l'acteur.

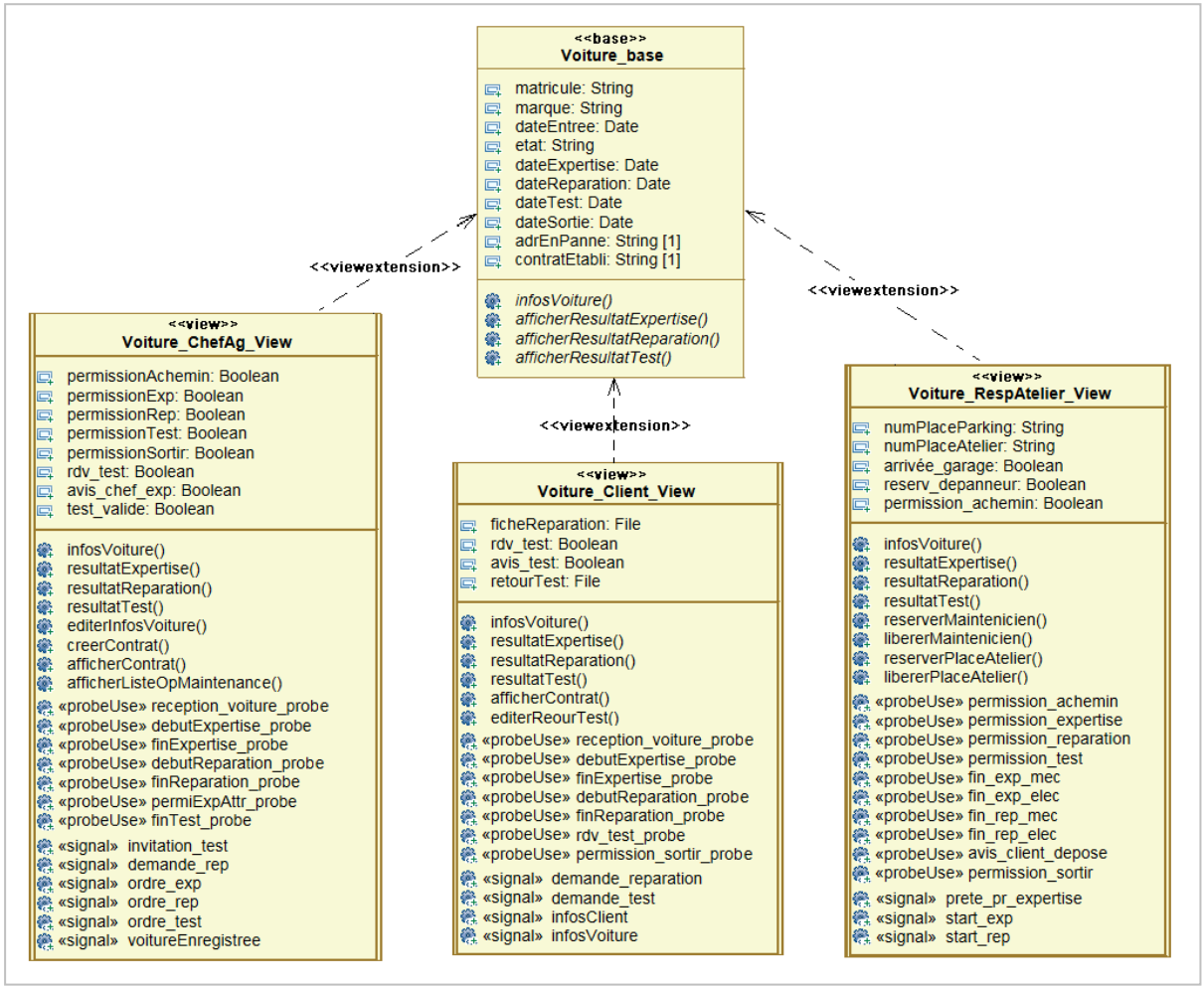


Figure V.13. Extrait du diagramme VxUML global : la classe multivue Voiture

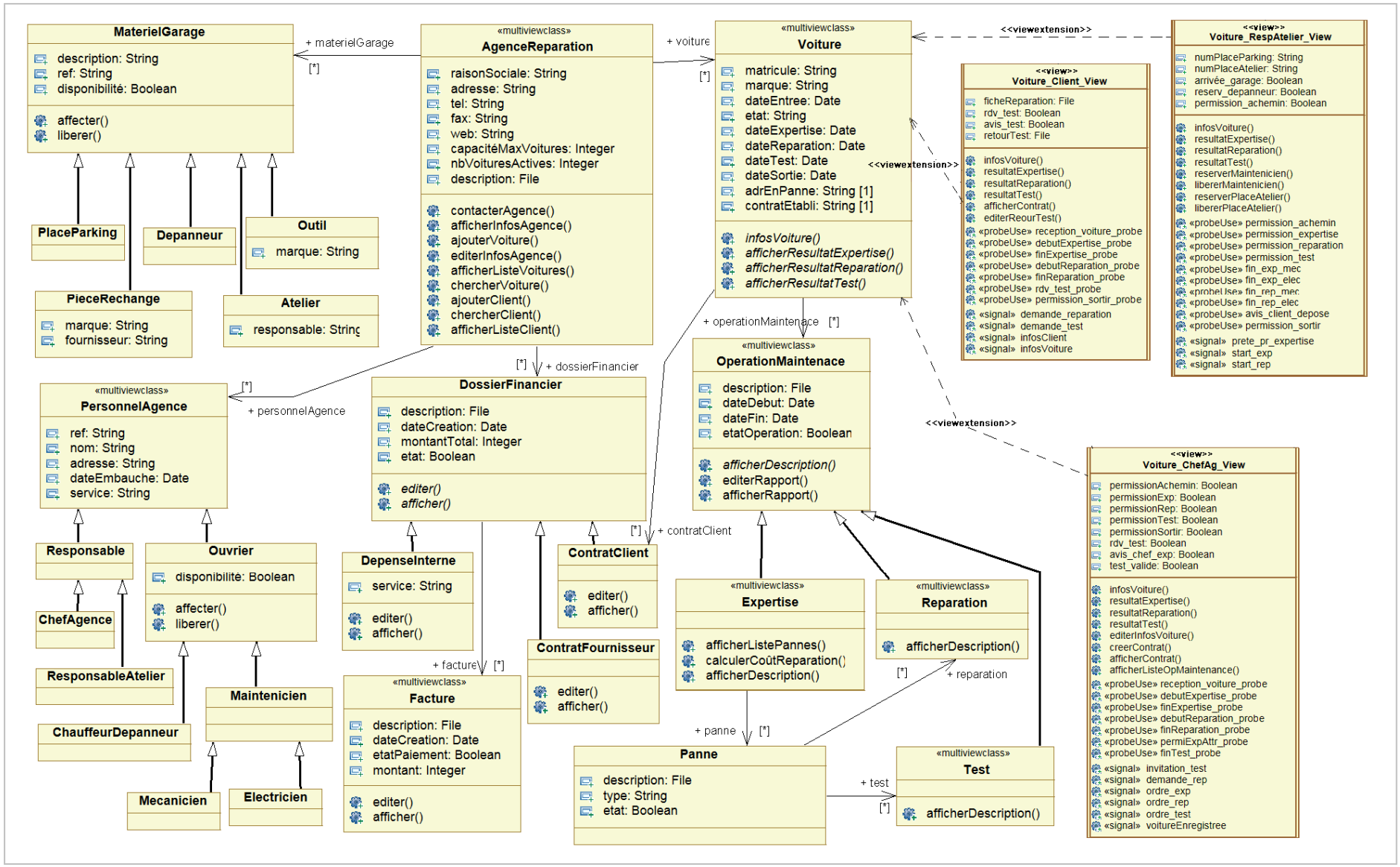


Figure V.14. Extrait du diagramme VUML obtenu par fusion des diagrammes de classes partiels des différents points de vue

V.3.4.2. Composition comportementale

La composition comportementale, effectuée après la fusion structurelle, consiste à composer le comportement des objets multivue (instances de classes multivue réactives) et à assurer la cohérence de l'ensemble. En fait, un objet multivue est composé d'un ensemble d'objets-vue ayant chacun un comportement décrit par une machine-vue développée (séparément) lors de la phase de conception par points de vue. Le comportement de l'objet multivue est simplement la composition parallèle des machines-vue. La mise en cohérence et les liens entre les machines-vue se fait à travers la concrétisation des sondes qu'elles utilisent, sondes jusque-là abstraites. Pour une sonde donnée, cette concrétisation se réalise par la définition des paramètres qui traduisent l'événement observé, décrites dans la note associée à la sonde. Cela ne modifie pas les modèles-vues de comportement car la déclaration des sondes est faite indépendamment de ces modèles.

Avant d'entamer cette phase de composition comportementale, on procède à une étape d'harmonisation et de résolution des éventuels conflits, comme dans la phase de fusion structurelle. Par exemple, il faut unifier les noms des sondes qui ont le même objectif, telles que *debut_expertise_probe* et *fin_expertise_probe* du point de vue *Client* avec les sondes *debutExpertise_probe* et *finExpertise_probe* du point de vue *ChefAgence*.

Dans cette section, nous donnons des exemples de définition de sondes abstraites déclarées dans la phase décentralisée. Il s'agit de : (1) la sonde *permission_sortir_probe* utilisée dans le point de vue *Client*, (2) la sonde *finExpertise_probe* utilisée dans le point de vue *Client*, (3) la sonde *reception_voiture_probe* utilisée dans le point de vue *ChefAgence* et (4) la sonde *debutExpertise_probe* utilisée dans le point de vue *ChefAgence*.

Exemple 1 : Sonde " permission_sortir_probe "

Dans la machine à états de la classe Voiture du point de vue *Client*, la transition de l'état "En_attente_sortie" vers l'état "Sortie" est franchie suite à l'action donnant la permission de livraison de la voiture au client. Or cette action ne fait pas partie des responsabilités du client ; en conséquence nous avons délégué cette tâche à la sonde *permission_sortir_probe*.

Dans cette phase de composition, les modèles-vue sont tous disponibles en entrée. Le développeur responsable de la composition revoit ces modèles et extrait l'événement qui représente la permission de livraison de la voiture. Il s'agit d'une décision du chef de l'agence exprimée par le changement de la valeur de l'attribut public "permission_sortir" de la classe Voiture_*ChefAgence*_View. La définition de la sonde *permission_sortir_probe* est illustrée par la Figure V.15.

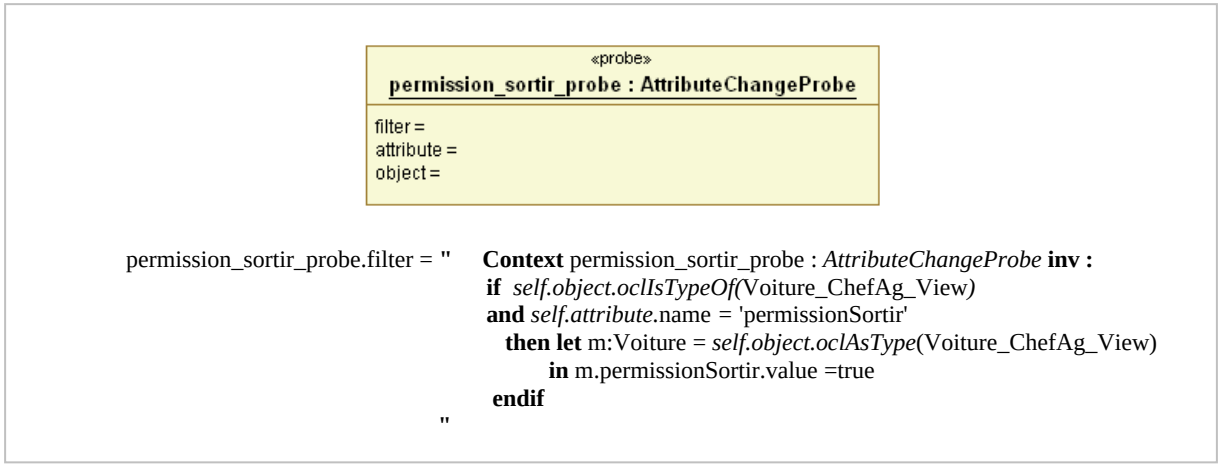


Figure V.15. Définition de la sonde "permission_sortir_probe" du point de vue Client

Exemple 2 : Sonde " finExpertise_probe "

Dans la machine-vue *Client* de la classe Voiture, la transition entre l'état "En_expertise" vers l'état "Reception_resultat_expertise" est franchie suite à l'action qui annonce la fin de l'expertise. Nous avons déclaré à cet effet la sonde *finExpertise_probe* pour recueillir cette information du système. Après consultation des différents modèles-vue, nous concluons que l'événement correspondant à cette action est la fin de l'exécution de la méthode editerRapportExp, appelé par la machine à état de Voiture_RespAtelier_View. La définition de la sonde *finExpertise_probe* est illustrée par la Figure V.16.

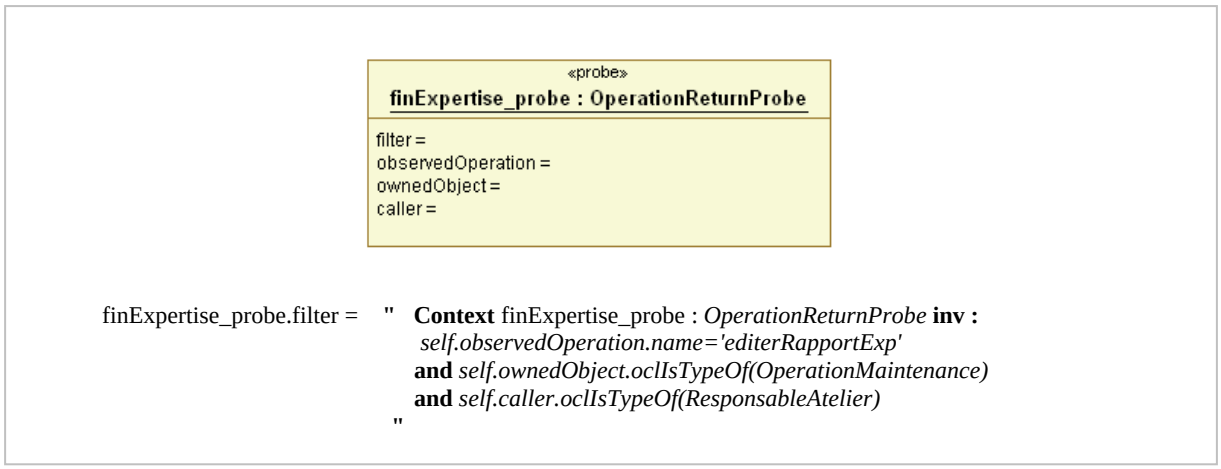
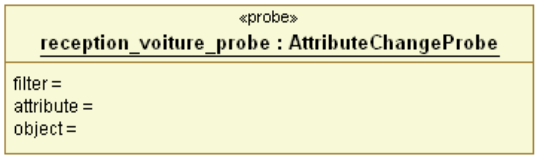


Figure V.16. Définition de la sonde "finExpertise_probe" du point de vue Client

Exemple 3 : Sonde " reception_voiture_probe "

Dans la machine-vue *ChefAgence* de la classe Voiture, la transition de l'état "Acheminement" vers l'état "Reception_garage" est franchie suite à l'action qui désigne l'arrivée de la voiture en panne dans le garage. L'objectif de la sonde *reception_voiture_probe* est de détecter cet événement. Après consultation des différents modèles-vue, nous concluons qu'il s'agit du changement de l'attribut *arrivee_garage* de la classe *Voiture_RespAtelier_View* qui reçoit la valeur *true*. La définition de la sonde *reception_voiture_probe* est illustrée par la Figure V.17.

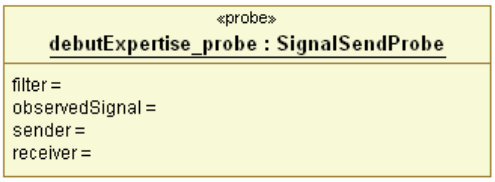



```
reception_voiture_probe.filter = "      Context reception_voiture_probe : AttributeChangeProbe inv :
                                        self.object.oclIsTypeOf(Voiture_RespAtelier_View)
                                        and  if  self.attribute.name = 'arrivee_garage'
                                                 then let ar:String = self.attribute.name
                                                      in ar.value =true
                                                 endif
                                 "
```

Figure V.17. Définition de la sonde "reception_voiture_probe" du point de vue ChefAgence

Exemple 4 : Sonde " debutExpertise_probe "

Dans la machine à états de la classe Voiture du point de vue *ChefAgence*, la transition de l'état "Attente_expertise" vers l'état "Expertise" est franchie suite à l'action qui annonce le début de l'expertise de la voiture. La sonde *debutExpertise_probe* a été déclarée pour détecter cet événement. Nous concluons après consultation des différents modèles-vue, qu'il s'agit de l'envoi du signal "startExp" par la classe *Voiture_RespAtelier_View*. La définition de la sonde *debutExpertise_probe* est illustrée par la Figure V.18.




```
debutExpertise_probe.filter = "      Context debutExpertise_probe : SignalSendProbe inv :
                                     self. observedSignal.oclIsTypeOf (startExp)
                                                and self.sender.oclIsTypeOf(Voiture_RespAtelier_View)
                              "
```

Figure V.18. Définition de la sonde "debutExpertise_probe" du point de vue ChefAgence

V.4. Conclusion

Nous avons présenté dans ce chapitre notre démarche de développement basée sur le profil VxUML et sa validation à travers une étude de cas. Nous avons tout d'abord donné une vue générale des trois phases de la démarche : la phase d'analyse globale, la phase de conception par points de vue et la phase de composition des modèles. Pour chaque phase, nous avons explicité les étapes nécessaires au traitement des aspects structurel et comportemental d'une modélisation multivue. La démarche associée à VxUML est en fait une extension de celle associée à VUML [Nassar, 05 ; Anwar, 09].

Le système "Gestion d'une agence de réparation de voiture" a constitué un cas d'étude significatif permettant d'appliquer et d'illustrer la démarche, et de montrer l'intérêt et la faisabilité des

concepts introduits dans le profil VxUML pour spécifier et composer le comportement dans une modélisation multivue.

Après l'application de l'approche à base de sondes sur ce cas d'étude, nous confirmons les affirmations énoncées dans la conclusion du chapitre précédent. L'approche à base de sondes s'adapte parfaitement aux systèmes de grande taille, et permet de surmonter le problème de la dépendance entre les vues.

Le développement de cette application a été basé sur le prototype que nous avons développé pour supporter la démarche VxUML. Le chapitre suivant présente ce prototype.

CHAPITRE VI. PROTOTYPE VXUML

Sommaire

VI.1. Introduction

La notion d'observation est un mécanisme puissant de spécification du comportement, mais qui nécessite une implémentation efficace pour être utile en pratique. Pour prouver la faisabilité des concepts et valider la démarche introduits dans les chapitres précédents, nous les avons mis en œuvre dans des outils support que nous avons regroupé dans un prototype opérationnel appelé VxUML. Ce chapitre décrit dans un premier temps l'implémentation de ce prototype, puis l'application de ce prototype au cas d'étude "Gestion d'une agence de réparation de voitures" présenté précédemment. L'outil a été développé dans un cadre IDM avec des techniques de transformation de modèles, dont nous reprenons ici les principes.

Pour l'implémentation du prototype, nous avons décidé d'utiliser la plateforme de développement open source Eclipse qui est la plus sollicitée par la communauté IDM. Ce choix est justifié par l'existence d'un grand nombre d'outils et de technologies autour de l'IDM tels que les API graphiques comme GEF (*Graphical Editing Framework*) [GEF, 07] et les technologies de gestion de modèles comme EMF (*Eclipse Modeling Framework*) [Budinsky et al., 03] et GMF (*Graphical Modeling Framework*) [GMF, 07]. Au niveau du langage de transformation de modèles, notre choix s'est porté sur le langage de transformation ATL [Jouault et al., 05]. C'est un langage créé pour s'inscrire dans une approche MDA, il est considéré maintenant comme un standard de transformation dans Eclipse et est intégré depuis 2007 dans le projet M2M [ATL, 07].

La suite du chapitre est organisée comme suit : la section VI.2 présente l'environnement de développement Eclipse et les outils associés à la gestion de modèles, à savoir le framework AMMA et le projet UML2. La section VI.3 détaille l'implémentation du prototype VxUML qui inclut le profil VUML structurel, le profil Probe_profile et la bibliothèque ProbeLibrary ; nous décrivons également le codage des règles de transformation en ATL, et la réalisation du vérificateur de modèles VxUML. La dernière section présente un exemple d'application de notre outil à travers la modélisation d'une partie de l'application "Gestion d'une agence de réparation de voitures" et donne également un aperçu du code Java généré.

VI.2. L'environnement de développement

Cette section présente les différentes technologies et outils utilisés pour réaliser notre prototype support à VxUML. Elle est décomposée en trois sous-sections : la première présente l'environnement de développement Eclipse ; la deuxième présente le framework AMMA associé à Eclipse pour la gestion de modèles ; la troisième concerne les techniques de transformation de modèles, notamment le langage ATL.

VI.2.1. L'environnement Eclipse

VI.2.1.1. Composantes d'Eclipse

Eclipse est une plateforme universelle qui offre un environnement de développement intégré et libre. Il est supervisé par le consortium Eclipse [Griffin, 04] composé d'une quarantaine de membres

dont IBM, Borland, Oracle, SAP, Telelogic et l'OMG. Son but est de fournir un environnement modulaire pour permettre de réaliser facilement des extensions. Le développement de nouvelles fonctionnalités se fait grâce à la notion de modules appelés plug-ins. Eclipse utilise intensément les plug-ins dans son architecture puisqu'en dehors du « Runtime », tout le reste est développé sous la forme de plug-ins (Figure VI.1).

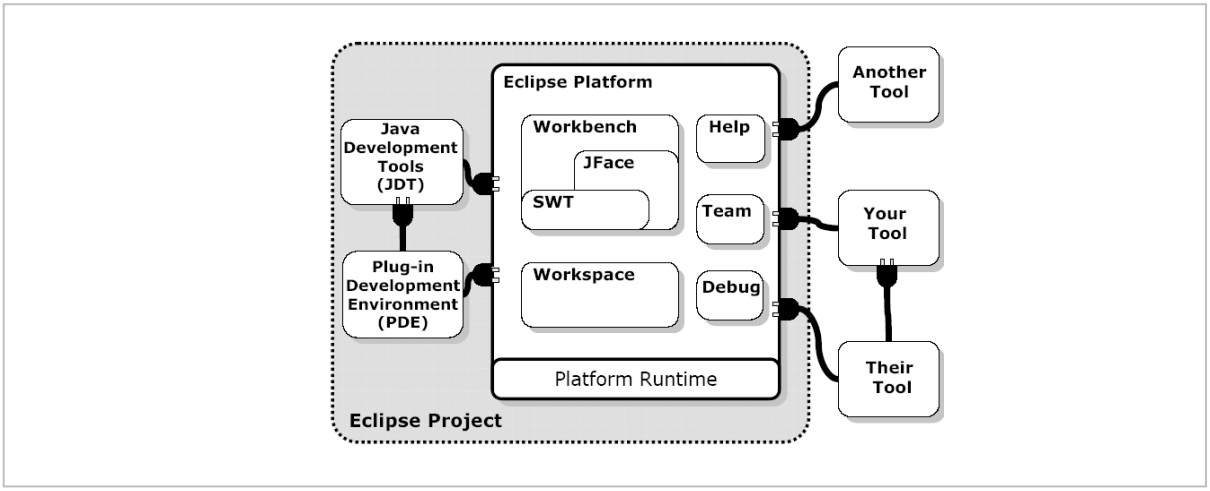


Figure VI.1. Architecture générale de la plateforme Eclipse [Griffin, 04]

La Figure VI.1 illustre l'architecture générale d'Eclipse. La plateforme définit un ensemble de frameworks et de services communs pour assurer une interopérabilité des outils ajoutés. C'est l'équivalent d'un noyau pour un système d'exploitation. Le workbench comprend un système de gestion concurrente de ressources distribuées, une gestion de projets, une gestion des versions, une infrastructure de débogage indépendante des langages de développement, un framework d'extension des fonctionnalités et une bibliothèque graphique portable (*Standard Widget Toolkit*, SWT). Le workspace comprend principalement un espace de stockage des ressources manipulées (.java, .class, .xml, etc.).

Après avoir présenté brièvement Eclipse et son architecture de base, nous allons voir dans les sous-sections qui suivent comment cette plateforme peut être utilisée pour réaliser des opérations de gestion de modèles comme l'édition et la transformation.

VI.2.1.2. Les modèles d'objets dans Eclipse

La plateforme Eclipse a été conçue sous forme de modèles d'objets qui sont représentés en Java [Griffin, 04]. Ces modèles jouent un rôle central en permettant l'intégration de différents outils. A titre d'exemple, le modèle de ressources représente l'espace de travail (workspace) ; il fournit une abstraction des fichiers et dossiers créés et organisés sous forme de projets. L'outil de développement Java JDT (*Java Development Tooling*) représente aussi une pièce importante dans cette architecture. En effet, le JDT intègre différentes représentations d'artefacts logiciels manipulés sous forme de code Java. Eclipse inclut aussi d'autres frameworks permettant la manipulation de modèles notamment EMF, basé sur un méta-modèle appelé Ecore. Enfin, Eclipse utilise aussi le standard XMI2.0 pour la sérialisation de modèles [OMG-XMI, 03].

Le framework EMF permet de décrire les modèles et fournit des supports à l'exécution de modèles, comme les notifications de modification, la persistance grâce à une sérialisation par défaut en XML ou XMI 2.0, ainsi qu'une API pour manipuler les objets EMF de façon générique. EMF a été principalement développé pour être utilisé dans les outils basés sur Eclipse. Initialement, il a été utilisé dans la suite de produits commerciaux *WebSphere Studio* d'IBM. Il est intégré dans les produits commerciaux comme l'outil Omondo EclispeUML [Omondo, 01]. De plus, ce framework est aussi utilisé dans de nombreux projets open source, tels que le framework de test Hyades [Hyades, 02] qui fournit une implémentation du profil UML de test de l'OMG.

Nous avons présenté dans les sections précédentes les fonctionnalités de base liées à la manipulation de modèles dans la plateforme Eclipse. Nous allons voir dans la suite comment ces fonctionnalités peuvent être étendues et utilisées afin de créer un environnement générique de gestion de modèles. Dans cette perspective, nous présentons la plateforme AMMA [Bézivin et al., 05b], sachant que d'autres projets de recherche ont conduit aussi au développement de plates-formes similaires comme Epsilon [Epsilon, 06], MIC [Sztipanovits et al., 97], etc.

VI.2.2. La plateforme AMMA

La plateforme AMMA [Bézivin et al., 05b] est un framework qui offre de nombreuses opérations de manipulation et de gestion de modèles telles que la transformation, le tissage et l'édition de modèles et de méta-modèles. Nous avons choisi AMMA comme plateforme pour implémenter notre approche d'une part pour ses fonctionnalités et son intégration à Eclipse, d'autre part car elle intègre le langage ATL qui a été choisi pour coder nos transformations. La Figure VI.2 présente l'architecture d'AMMA basée sur EMF. La fonction de transformation dans AMMA est assurée par le langage ATL. Un IDE (*Integrated Development Environment*) et un moteur d'exécution ont été également développés pour ce langage sous forme de plateforme Eclipse. Cet environnement facilite le processus d'écriture de transformations grâce à un éditeur de code, un débogueur et un schéma présentant une vue de la syntaxe abstraite de la transformation du programme en cours d'édition.

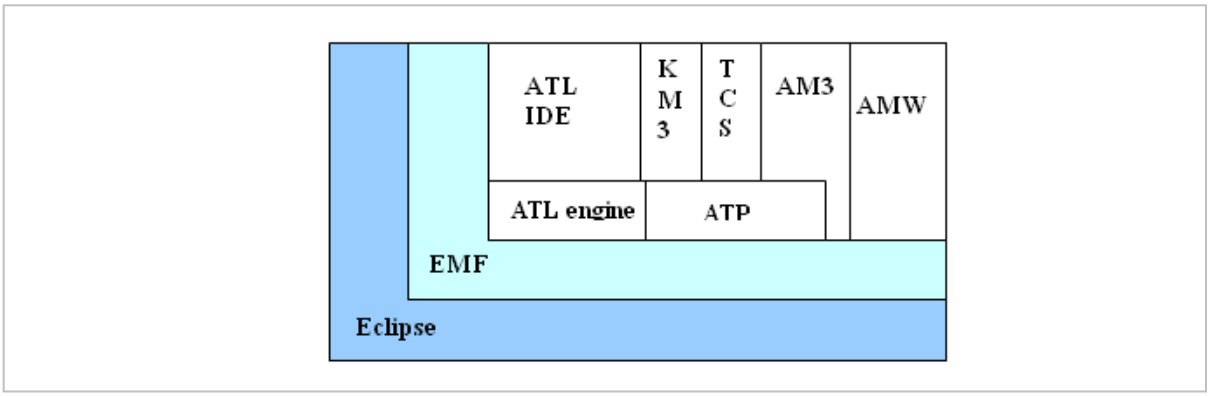


Figure VI.2. Architecture de la plateforme AMMA

Nous présentons ci-dessous les caractéristiques principales des composants d'AMMA :

- ATL est un langage dédié de transformation de modèles. Il utilise des méta-modèles source et cible pour définir une transformation de modèle. En dehors des transformations, il peut être utilisé

également pour la vérification de modèles [Bézivin et al., 05]. Nous donnons plus de détails sur ce langage dans la section VI.2.3.2.

- KM3 est un langage de définition de méta-modèles basé sur les concepts du MOF 2.0 [OMG-MOF] et EMF/Ecore [Budinsky et al., 03]. La syntaxe concrète de KM3 est actuellement basée sur une notation textuelle proche du langage Java.

- L'outil AMW permet de définir des liens de tissage entre les éléments de deux ou plusieurs modèles ; ces liens sont définis dans un modèle de tissage qui a pour objectif principal la représentation des différents types de liens entre les éléments des modèles.

- L'outil de gestion de méga-modèles AM3 fournit un support de modélisation et de coordination globale entre les ressources dans un développement orienté IDM.

- Le langage TCS [Jouault et al., 06c] est un langage de transformation de programmes en modèles et de modèles en programmes. Il sert à représenter syntaxiquement les concepts classiques d'un langage (mots clé, symboles, structures de contrôle et d'ordonnancement, etc. En utilisant ce langage, les modèles peuvent être sérialisés en texte et les fichiers textes peuvent être analysés afin de générer les modèles correspondants.

- L'outil ATP définit une série d'injecteurs/extracteurs permettant d'assurer les opérations d'importation et/ou d'exportation de modèles venant d'espaces technologiques différents (classes Java, modèles relationnels, etc.).

VI.2.3. Transformation de modèles

La transformation de modèle est une opération fondamentale dans toute approche orientée modèle. La définition et l'automatisation des transformations a pour objectif de rendre les modèles plus opérationnels et d'augmenter la productivité du développement. Dans cette section nous présentons dans un premier temps le principe général de la technique de transformation, puis le langage de transformation ATL que nous avons choisi pour coder les transformations au sein de notre prototype.

VI.2.3.1. Principe général d'une transformation

Une transformation de modèle se définit par l'opération de génération d'un ou plusieurs modèles cible à partir d'un ou de plusieurs modèles source, conformément à une définition de transformation [Bézivin, 04]. L'OMG a lancé en 2002 un appel à proposition industriel RFP MOF/QVT [OMG-QVT] dont le but était de définir un langage unifié de transformation. L'idée est de modéliser la transformation elle-même en lui appliquant les principes de l'IDM, et donc de la considérer comme une application définie par un modèle de transformation (Mt) lui-même conforme à un méta-modèle (MMt), ce dernier représentant la définition abstraite du langage de transformation [Bézivin, 04] (voir la Figure VI.3).

Cette approche de définition de transformation est dite par modélisation ou par méta-modélisation, par opposition aux autres approches telles que l'approche par programmation utilisant

des API de manipulation de modèles comme JMI ou EMF [Budinsky et al., 03], ou l'approche par template utilisée généralement pour les transformations modèle-vers-texte. En se basant sur ces principes, une transformation de modèle définie par un modèle Mt est l'opération qui permet de transformer un modèle Ma (conforme à son méta-modèle MMa) en un modèle Mb (conforme à son méta-modèle MMb).

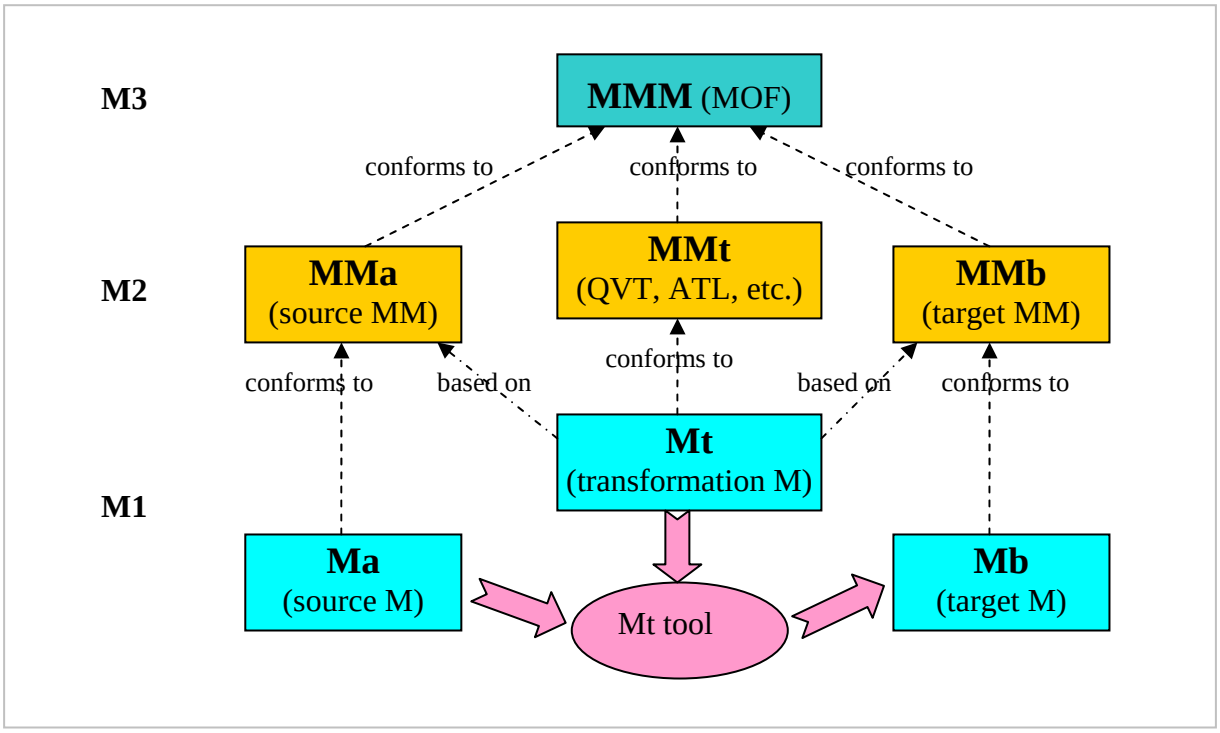


Figure VI.3. Approche de transformation basée sur les méta-modèles

Les approches permettant de réaliser des transformations de modèle selon la technique par méta-modélisation sont nombreuses. Généralement, elles appliquent les techniques de l'IDM aux transformations elles-mêmes. Le principe est d'offrir un méta-modèle permettant de construire des modèles de transformation. Dans cet esprit, plusieurs langages de transformation ont été proposés, parmi lesquels le standard MOF2.0 QVT [OMG-QVT], le langage ATL [Jouault et al., 05], et le langage Kermeta [Muller et al., 05].

Notre choix s'est porté sur le langage de transformation ATL. C'est un langage créé pour s'inscrire dans une approche MDA, il est considéré maintenant comme un standard de transformation dans Eclipse et est intégré depuis 2007 dans le projet M2M [ATL, 07]

VI.2.3.2. Le langage ATL

ATL fait partie des langages de transformation, c'est un langage hybride qui permet de définir des constructions déclaratives et impératives. ATL s'appuie sur une architecture à trois niveaux. Au premier niveau, on trouve la machine virtuelle d'ATL dont le rôle est d'exécuter les programmes ATL compilés écrits au deuxième niveau [Jouault et al., 06a]. Le niveau supérieur de cette architecture est défini par AMW [Del Fabro et al., 06] qui peut être utilisé optionnellement comme langage de spécification de transformation à un niveau d'abstraction élevé.

Au niveau langage, ATL est défini par une syntaxe concrète textuelle et une syntaxe abstraite définie par un méta-modèle conforme au MOF. Une règle déclarative d'ATL est appelée *matched rule* ; elle est spécifiée par un nom, un ensemble de patrons source (*InPattern*) qui sont associés avec les éléments source, et un ensemble de patrons cible (*OutPattern*) représentant les éléments à créer dans le modèle cible. Le style impératif d'ATL est supporté par deux constructions différentes. En effet, on peut utiliser soit des règles impératives appelées *called rule*, soit un bloc d'instructions impératives (*ActionBlock*) utilisé avec les deux types de règles. Une *called rule* est appelée explicitement en utilisant son nom et en initialisant ses paramètres. La Figure VI.4 présente un extrait de la syntaxe abstraite des règles de transformation ATL.

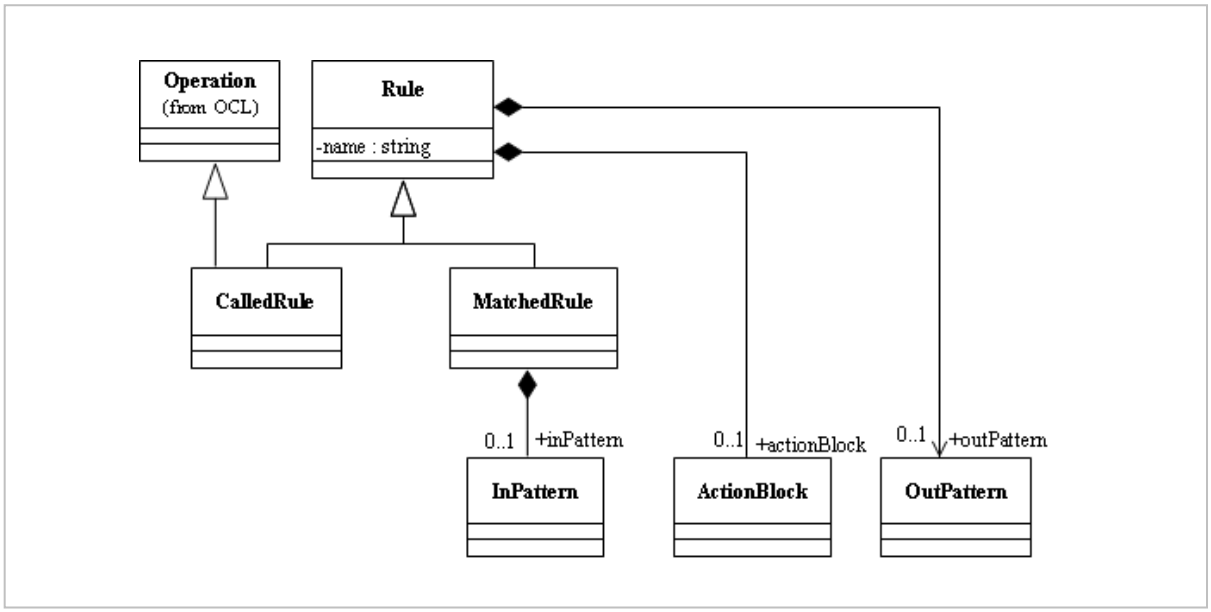


Figure VI.4. Syntaxe abstraite des règles de transformation ATL [Jouault et al., 06a]

ATL supporte deux modes d'exécution : standard et raffinement. Dans le mode standard, les éléments sont créés seulement quand les patrons source définis dans les règles déclaratives ont été reconnus dans le modèle source. Le système instancie alors les éléments des patrons cible. Une fois l'étape d'instanciation passée, un lien de traçabilité est créé associant chaque élément reconnu du modèle source à un élément créé du modèle cible. Finalement, le système évalue ces liens de traçabilité afin de déterminer les valeurs des propriétés des éléments instanciés. Dans le mode raffinement, par opposition au mode standard, les éléments dont les patrons source n'ont pas été associés par les règles sont automatiquement copiés dans le modèle cible par le moteur d'exécution.

Au niveau implémentation, un prototype de moteur d'exécution pour le langage ATL a été développé et est disponible à travers le projet M2M d'Eclipse [Eclipse, 07]. Ce moteur d'exécution est basé sur une machine virtuelle de transformation [Jouault, 06].

VI.3. Implémentation du prototype VxUML

Cette section présente notre prototype support à VxUML. Elle est composée de deux sous-sections : la première donne un aperçu général de l'architecture du prototype ; la deuxième présente les différents modules constituant l'architecture proposée.

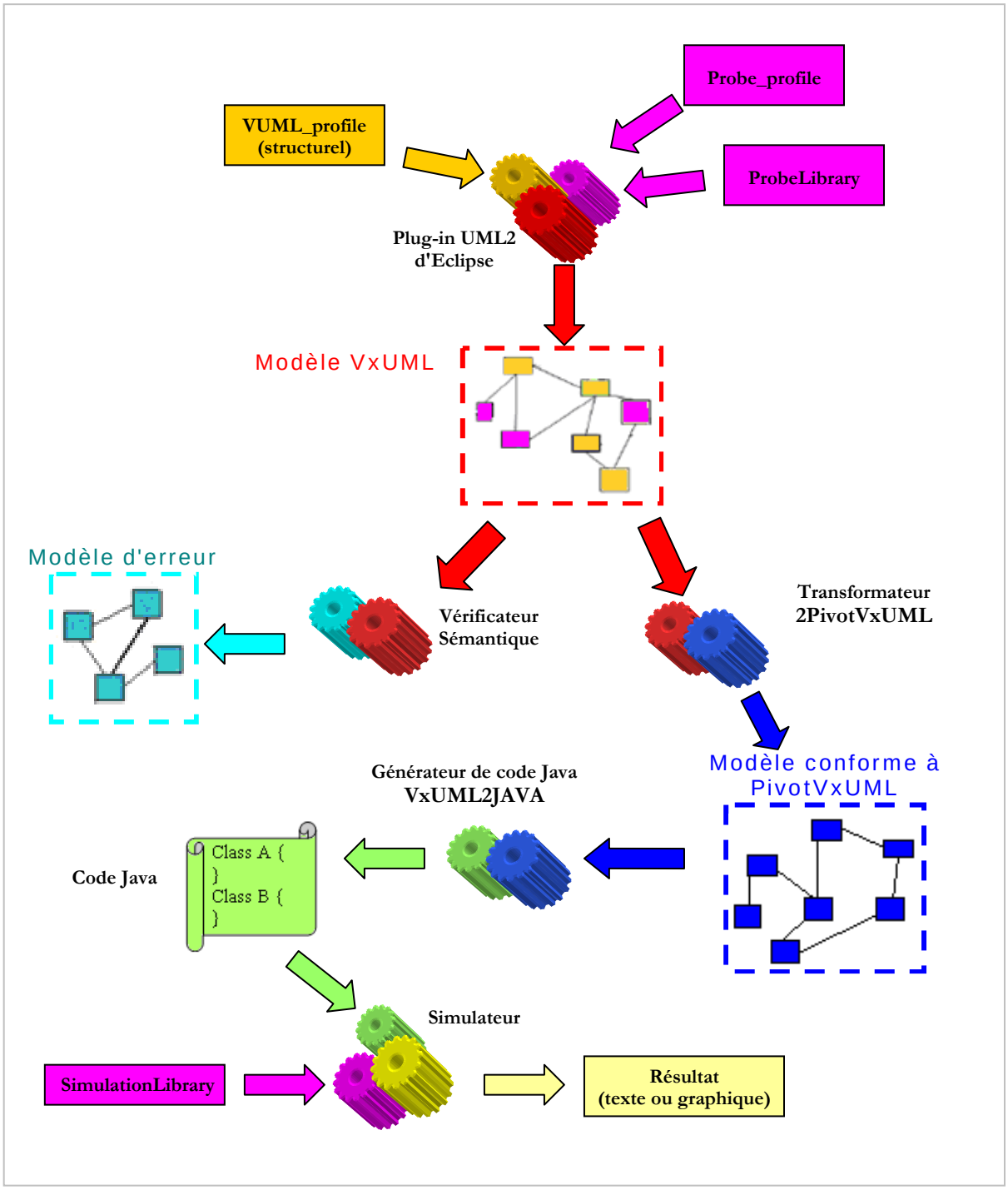


Figure VI.5. Architecture générale du prototype VxUML

VI.3.1. Architecture générale

Notre prototype VxUML fournit une plateforme intégrée composée de plusieurs modules (cf. Figure VI.5). Le premier module sert à la spécification des modèles VxUML. Le deuxième module est un vérificateur sémantique de modèles VxUML. Le troisième est un transformateur ATL,

2PivotVxUML. Le quatrième module est un générateur de code Java. Et à la fin on trouve un module qui sert à la simulation du code Java généré.

Les éléments principaux du prototype sont les suivants :

- **Les profils associés à VxUML**. (1) le profil *"VUML_profile"* représente les concepts structurels de notre approche, à savoir, les notions de *"base"*, *"view"* et de *"viewExtension"* ; (2) le profile *"Probe_profile"* représente les concepts comportementaux relatifs à la notion de sondes d'événements, à savoir, *"probe"*, *"probeUse"* et *"wait"* ; et (3) la bibliothèque *"ProbeLibrary"* qui fournit à l'utilisateur un ensemble de classes prédéfinies de sondes.
- **Le vérificateur de sémantique de VxUML**. Il vérifie la cohérence des modèles VxUML. Il est basé sur la sémantique de VUML structurel définie dans [Nassar, 05 ; Anwar, 09] ainsi que sur la sémantique associée au profil Probe_profile décrite dans la section IV.3.3.2.
- **Le transformateur 2PivotVxUML**. Il prend en entrée un modèle VxUML et génère un modèle simplifié de VxUML. Il s'agit d'un modèle pivot où les concepts complexes manipulés dans le modèle VxUML sont traduits aux éléments simples conformes aux éléments usuels dans la programmation orientée objet (classe, objet, attribut, etc.).
- **Le générateur de code Java**. Ce générateur permet de produire du code Java à partir d'un modèle VxUML simplifié. Le résultat final du processus est un ensemble de fichiers texte (.java) représentant le code objet de l'application.
- **Le simulateur**. Son rôle est de simuler l'exécution du code Java généré. Le simulateur se base, en plus du code Java généré, sur une librairie de classes support et de scénarios élémentaires pour la simulation des sondes.

Dans la suite de cette section, nous donnons quelques détails sur l'implémentation de chaque module de notre prototype.

VI.3.2. Implémentation des modules du prototype

Pour modéliser les profils associés au prototype VxUML, nous avons utilisé le plug-in UML2 d'Eclipse. Ce plug-in utilise le framework EMF pour accéder à l'implémentation du méta-modèle standard UML2.0, et permet de créer des profils UML correspondant à ce méta-modèle.

Concernant le vérificateur, le transformateur 2PivotVxUML et le générateur de code Java, nous utilisons les techniques de transformation de modèles avec le langage ATL. Pour l'implémentation du simulateur, elle est basée sur le compilateur Java.

VI.3.2.1. Module de spécification des modèles VxUML

Ce module se sert des deux profils VUML_profile et Probe_profile ainsi que de la bibliothèque ProbeLibrary pour donner au concepteur le moyen de réaliser son projet suivant la démarche VxUML.

Le profil VUML_profile structurel

Nous rappelons tout d'abord que le profil VUML présenté dans le Chapitre III section III.4.3, a pour objectif de spécialiser UML afin de supporter la modélisation par vues/points de vue. L'ajout principal à UML est le concept de classe multivue. Pour modéliser avec ce concept, le profil VUML_profile, conforme à UML 2.0, a introduit un certain nombre de stéréotypes, à savoir : « *base* », « *view* », « *abstractView* », « *viewExtension* », « *multiViewsClass* » et « *viewDependency* ». La sémantique de chacun d'entre eux a été détaillée dans [Nassar, 05]. Nous n'implémentons dans notre prototype que les stéréotypes indispensables « *base* », « *view* » et « *viewExtension* ». La Figure VI.6 donne les extensions des méta-classes associées à ces trois stéréotypes. Les stéréotypes « *base* » et « *view* » étendent la méta-classe *Class* et le stéréotype « *viewExtension* » est défini comme une extension de la méta-classe *Dependency*.

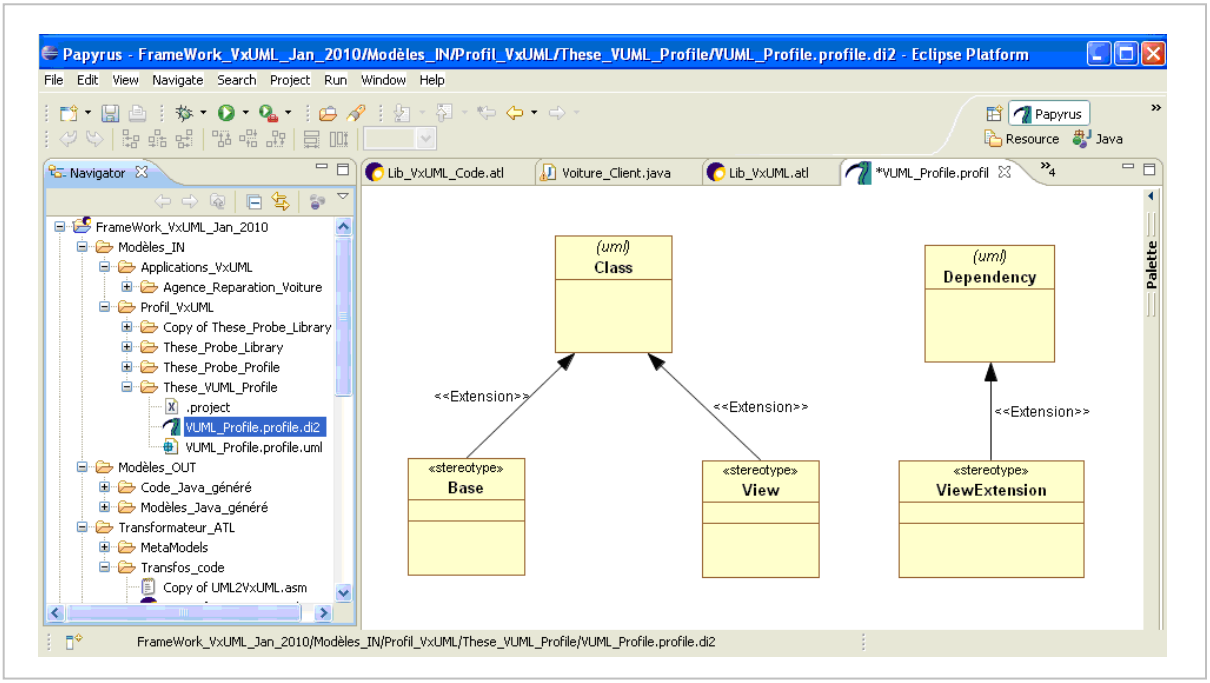


Figure VI.6. Stéréotypes associés au profil VUML_profile

Le profil Probe_profile

Le profil Probe_profile que nous avons détaillé dans la section IV.3.3.2 introduit un certain nombre de stéréotypes pour représenter le concept d'observation d'événements et les éléments associés. Ces stéréotypes sont : *Probe*, *ProbeClass*, *ProbeUse*, *ProbeEvent*, et *Wait*. Ces stéréotypes sont le résultat de l'opération du mapping à partir du méta-modèle présenté dans la section IV.3.3.2.1.

La bibliothèque ProbeLibrary

Nous rappelons que la bibliothèque ProbeLibrary est présentée dans la section IV.3.2.1. Les sondes d'événements sont de trois types : (i) les sondes d'événements de communication ; (ii) les sondes d'événements liés aux changements dans la structure du système ; et (iii) les sondes de modification de données. Ces types de sondes élémentaires se situent au niveau de modélisation M1 offrant ainsi aux concepteurs la possibilité de les utiliser dans les modèles de conception comme des classes prédéfinies. Chaque type de cette bibliothèque peut être instancié et devient un élément du

système qui peut être adapté aux besoins de l'application, par projection, ou par dérivation ou bien par composition comme décrit dans le chapitre IV dans les sections IV.3.2.4, IV.3.2.5 et IV.3.2.6.

VI.3.2.2. Vérificateur de sémantique de VxUML

Le vérificateur de modèles VxUML implémente la sémantique statique et notamment les règles de bonne formation (*well formedness rules*) de VxUML. Ces règles proviennent de la sémantique de VUML structurel décrite dans [Nassar, 05] ainsi que de la sémantique associée au profil Probe_profile définie dans la section IV.3.3.2. La Figure VI.7 décrit le principe du vérificateur ATL du modèle VxUML.

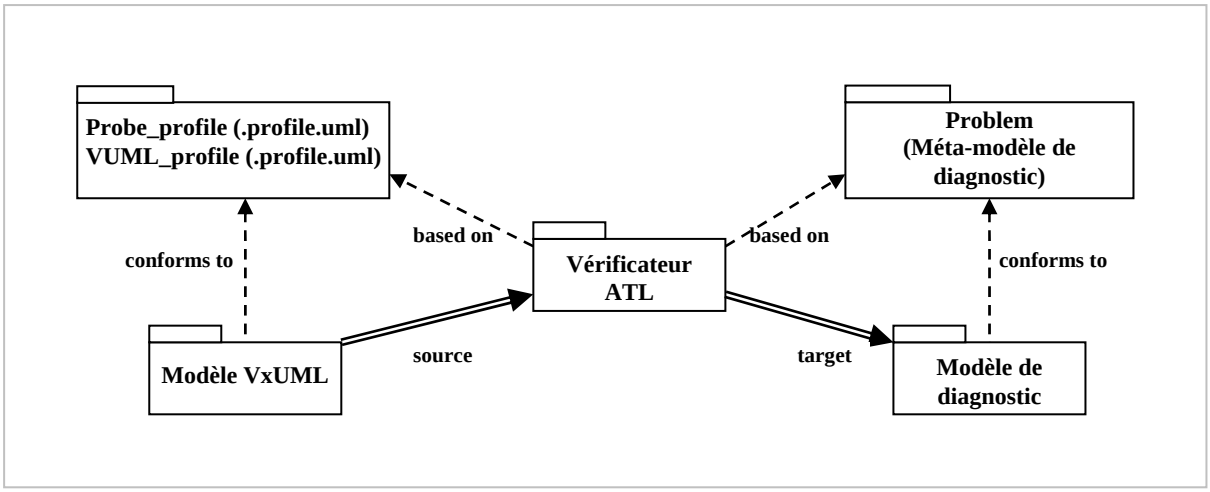


Figure VI.7. Principe de la vérification du modèle VxUML avec ATL

La technique de vérification utilisée est basée sur l'utilisation du langage ATL et suit le principe de transformation de modèles présenté dans la section VI.2.3.1 ci-dessus. Cette transformation (le vérificateur) permet de générer un modèle de diagnostic suivant le principe décrit dans [Bézivin et al., 05a]. Le méta-modèle utilisé est Problem présenté sur la Figure VI.8. Le modèle de diagnostic est le résultat de l'opération de vérification d'un ensemble de contraintes par le modèle source (modèle VxUML dans notre cas). Le modèle de diagnostic généré peut être personnalisé (sous format textuel ou graphique) par la suite, pour offrir plus de détail sur les erreurs identifiées. Le principe de cette approche est de définir une règle ATL pour chaque contrainte à vérifier. Le contexte de la contrainte OCL devient l'élément source de la règle ATL, tandis que la condition de garde de la règle est la négation de l'expression booléenne associée à la contrainte. Finalement, le patron de sortie de la règle est un type qui définit une erreur (Problem). Ce type, défini dans le méta-modèle de diagnostic (cf. Figure VI.8), offre des précisions sur l'erreur (sévérité, localisation, description, etc.).

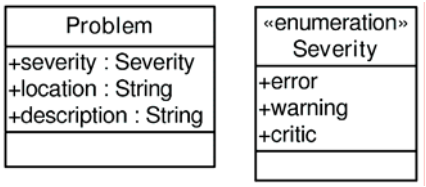


Figure VI.8. Le méta-modèle de diagnostic : Problem [Bézivin et al., 05a]

Nous présentons ici l'exemple de deux règles ATL qui vérifient deux contraintes. La première contrainte est associée au profil VUML_profile et la deuxième est associée au profil Probe_profile. Nous avons présenté les deux contraintes OCL à vérifier sous forme de helpers ATL. Nous rappelons que les helpers ATL sont équivalents aux fonctions globales qu'on retrouve dans certains langages de programmation. Ils servent à calculer une information utilisable dans le module par les règles de transformation ou par d'autres helpers en spécifiant des opérations de navigation sur les modèles source. Les helpers peuvent avoir des paramètres, et ils utilisent des expressions OCL pour calculer la valeur de retour.

La Figure VI.9 montre le code de la règle *BaseRule1* ainsi que le code du helper *BaseCondition1* spécifiant la contrainte vérifiée par cette règle. L'objectif de cette règle est d'assurer qu'une classe stéréotypée par « *base* », a au moins une relation « *viewExtension* ».

```
rule BaseRule1 extends printNameElement{
    from c : VxUML2!Class( not c.BaseCondition1)
    to p : Problem!Problem
        (
        severity <-#error,
        description <- 'the class ' + c.name + 'must have at least one viewExtension relationship' +' or must
             be a child of a base class'
        )
    do {    p.description.output();          }
}

helper context VxUML2!Class def : BaseCondition1 : Boolean =
    if thisModule.inElements->includes(self) and self.hasStereotype('base') then
        if not self.getViewExtensions().oclIsUndefined() and not self.getViewExtensions()->isEmpty()
            then self.getViewExtensions()->select(d | d.hasStereotype('viewExtension'))->size() >=1 or
                 self.isStereokinded('base')
        else true
        endif
    else  true
    endif
    ;
```

Figure VI.9. Exemple de règle ATL pour la vérification de la sémantique de VUML_profile

La Figure VI.10 montre le code de la règle *ProbeRule1* ainsi que le code du helper *ProbeCondition1* spécifiant la contrainte vérifiée par cette règle. L'objectif de cette règle est d'assurer qu'un élément de

type InstanceSpecification stéréotypé par *« probe »*, ne peut être qu'instance d'une classe stéréotypée *« probeClass »*.

```
rule ProbeRule1 extends printNameElement
{   from i : VxUML2!InstanceSpecification(     not i.ProbeCondition1)
    to p : Problem!Problem
        (
        severity <-#error,
        description <- 'the instance ' + i.name + 'must have as classifier with « probeClass » stereotype'
        )
    do {    p.description.output();          }
}

helper context VxUML2!InstanceSpecification def : ProbeCondition1 : Boolean =
    if thisModule.inElements->includes(self) and self.hasStereotype('probe') then
        if not self.classifier.oclIsUndefined()
            then self.classifier->first().hasStereotype('probeClass')
        else true
        endif
    else  true
    endif
    ;
```

Figure VI.10. Exemple de règle ATL pour la vérification de la sémantique de Probe_profile

Nous avons donné dans l'annexe B une vision plus élargie des règles de vérification développées pour cerner les concepts manipulés dans notre prototype.

VI.3.2.3. Transformateur de modèle 2PivotVxUML

Le transformateur 2PivotVxUML prend en entrée un modèle VxUML et génère en sortie un modèle simplifié de ce dernier. En fait, notre objectif est de réaliser une plateforme intégrée capable de réunir des outils divers pour la manipulation de modèles VxUML allant d'un éditeur de modèle VxUML jusqu'à la génération de code ciblant les langages objet reconnus, et également offrir des outils de simulation adaptés à ces langages de programmation. L'objectif du modèle PivotVxUML est de donner une représentation simplifiée des concepts spécifiques du domaine de la modélisation multivue sans passer par les éléments de profil UML de VxUML (stéréotypes et *tagged values*) qui alourdissent la description.

Nous nous sommes basé pour développer le méta-modèle PivotVxUML sur une version simplifiée du méta-modèle de Java [Java-meta, 05]. Ce dernier a été étendu avec des versions simplifiées des éléments en provenance du méta-modèle de VxUML. Ceci permet une navigation plus aisée dans un modèle multi-vue. Par exemple au lieu de lancer à chaque fois des opérations de sélection dans un modèle VxUML pour recueillir, en naviguant sur les dépendances, les vues associées à une base, cette opération est réalisée une seule fois dans le cadre de la trasformation 2PivotVxUML et le résultat est stocké dans une liste de vues au sein de la base.

Le modèle généré sera utilisé par la suite comme base pour les futures opérations telles que la génération de code multi-cibles (nous avons ciblé Java dans un premier temps), la simulation et le test. Nous donnons en annexe A le méta-modèle PivotVxUML développé au format KM3

```
module 2PivotVxUML; -- Module Template
create OUT : PivotVxUML from IN : UML2, VUML_profile : UML2, Probe_profile : UML2,
ProbeLibrary : UML2;

rule toPivotVxUMLClass
      {  from c :UML2!Class(c.oclIsTypeOf(UML2!Class) )
         to a : PivotVxUML!Class
         (    name<-c.name,
              isAbstract<-c.isAbstract,
              isPublic<-c.isPublic(),
              extend<-c.getParent(),
              owner<-c.getPathPackage(),
              ownedBehavior<-c.ownedBehavior,
              isView<-c.isView(),
              isBase<-c.isBase(),
              base<-c.getBase(),
              isProbeClass<-c.isProbeClass(),
         )
      }
rule toPivotVxUMLObjet
      {  from c :UML2!InstanceSpecification
         to a : PivotVxUML!Object
         (    name<-c.name,
              owner<-c.namespace,
              instanceOf<-c.classifier,
              isProbe<-c.isProbe()
         )
      }
helper context UML2!Namespace def : getPathPackage() : String =
    if self.namespace.oclIsUndefined() then      ''
    else if self.namespace.oclIsTypeOf(UML2!Model) then    ''
    else    self.namespace.getPathPackage() + '.'
    endif endif + self.name
    ;
helper context UML2!Class def : getBase() : UML2!Class =    -- récupération de la base de la vue dans B1
    let vex : UML2!Dependency= self.clientDependency->select(r | r.hasStereotype('ViewExtension'))->
          asSequence()
        in if not vex.oclIsUndefined() and not vex->isEmpty()
                then vex->first().supplier->first()
                else    OclUndefined
            endif
    ;
helper context UML2!Class def : isBase() : Boolean = self.hasStereotype('Base') ;
helper context UML2!Class def : isView() : Boolean = self.hasStereotype('View') ;
helper context UML2!InstanceSpecification def : isProbe() : Boolean =  self.hasStereotype('Probe') ;
helper context UML2!Class def : isProbeClass() : Boolean =self.hasStereotype('ProbeClass') ;
```

Figure VI.11. Extrait des règles ATL pour la transformation 2PivotVxUML

La Figure VI.11 donne un extrait de deux règles qui transforment, respectivement, les éléments de type Class et InstanceSpecification se trouvant dans le modèle VxUML vers des classes et objets du modèle PivotVxUML. Dans la section entête du module on trouve le nom du module, la

déclaration des modèles source, cible, avec leurs méta-modèles correspondants. Sur cet exemple, les modèles IN, VUML_profile, Probe_profile et ProbeLibrary représentent les modèles source qui sont conformes au méta-modèle UML2. Le modèle OUT dénote le modèle de sortie à générer qui est conforme au méta-modèle PivotVxUML. Pour une lecture plus aisée pour ces deux règles ATL, nous rappelons qu'une définition de transformation ATL est constituée de quatre sections : une section entête, une section d'importation de bibliothèques, une section de déclaration des 'helpers' et une section de spécification des règles de transformation. Les règles de transformation sont déclarées sous forme de *Matched Rules* (la forme déclarative d'ATL) ; leurs invocations se fait implicitement par le moteur ATL. La règle toPivotVxUMLClass, par exemple, produit en sortie une classe PivotVxUML à partir d'une classe UML.

Une fois le modèle PivotVxUML généré, l'étape suivante se focalise sur la génération du code objet. Actuellement notre prototype génère du code Java, et nous comptons l'étendre pour cibler d'autres langages comme C++ par exemple. Le principe de génération de code Java est présenté dans la section suivante.

VI.3.2.4. Générateur de code Java

Dans un premier temps, nous avons choisi de cibler le langage Java [Java-meta, 05]. Mais les techniques utilisées dans cette génération de code sont réutilisables pour cibler d'autres langages objet (C++, Eiffel, Ruby, …). La Figure VI.12 présente l'architecture générale du module de génération de code ; ce module prend en entrée un modèle PivotVxUML, (son méta-modèle en KM3 est présenté dans l'annexe A) et génère en sortie un ensemble de fichiers ".java".

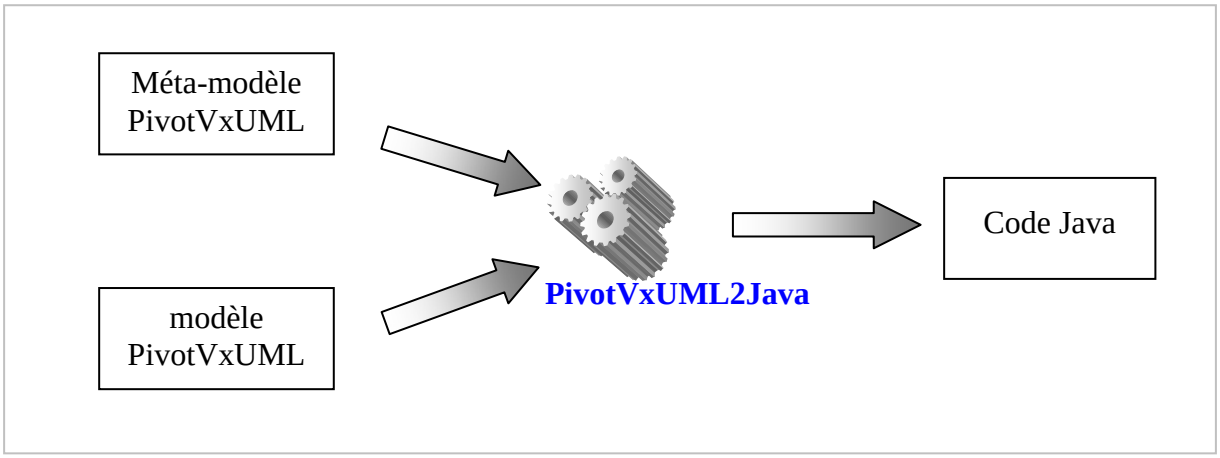


Figure VI.12. Schéma général de la transformation d'un modèle PivotVxUML en code objet Java

La requête ATL que nous avons développée pour générer le code est présentée dans la Figure VI.13 ci-dessous.

```
query PivotVxUML2Java = PivotVxUML! PivotVxUMLElement.allInstances()->collect(c |
        if c.oclIsTypeOf(PivotVxUML!Class) then
            if c.ownedBehavior->first().oclIsUndefined()
                then c.toString().writeTo('C:/ATL_UML2_Bundle_2.0.0RC2_Windows/eclipse/workspace/
                        Framwork_VxUML_Juin_2009/Modeles_IN-
                        OUT/3_Code_Java_Généré/src/application/'
                        + c.owner.name + '/'+ c.name + '.java')
            else c.toString().writeTo('C:/ATL_UML2_Bundle_2.0.0RC2_Windows/eclipse/workspace/
                        Framwork_VxUML_Juin_2009/Modeles_IN-
                        OUT/3_Code_Java_Généré/src/application/'
                            + c.owner.name + '/'+ c.name + '.java')
                        and c.generateModelClasses()
                        and c.generateApplicationClasses()
            endif
        else if c.oclIsTypeOf(PivotVxUML!Signal)
            then    c.toString().writeTo('C:/ATL_UML2_Bundle_2.0.0RC2_Windows/eclipse/workspace/
                        Framwork_VxUML_Juin_2009/Modeles_IN-
                        OUT/3_Code_Java_Généré/src/application/signalList/'
                        + c.name + '.java')
            else ''
            endif
        endif
    )
    ;

uses Lib_ PivotVxUML2Java;
uses Lib_ PivotVxUML2Java _Code;
```

Figure VI.13. Extrait de la transformation PivotVxUML2Java

Cette requête utilise deux bibliothèques Lib_PivotVxUML2Java et Lib_PivotVxUML2Java_Code qui représentent le patron de génération écrit en ATL. La requête permet de naviguer dans le modèle source PivotVxUML et de sélectionner les éléments instances de Class dont le contenu sera édité dans un ficher texte (.java). Bien entendu, la définition de la méthode toString sur la classe engendre aussi l'invocation de cette méthode sur tous les éléments de type propriété et méthode appartenant à cette classe. La définition de cette méthode est spécifiée dans la bibliothèque que nous avons développée et nommée Lib_PivotVxUML2Java_Code. La Figure VI.14 donne un extrait de cette bibliothèque.

```
library PivotVxUML2Java_Code;

helper context PivotVxUML!Class def : libImport() : String =
'\n\nimport java.util.List;
import java.util.ArrayList;
import model.*;
import application.signalList.*;'
    ;

helper context PivotVxUML!Class def : creatSM() : String =
    '\n\npublic void createSM()'
    +'\n    {'
    +'\n    Signal sig=new Signal();'
    + if self.ownedBehavior->first().region->.transition->notEmpty() then
    +'Transition tr =new Transition();'
    +self.ownedBehavior->first().region->iterate(reg; acc : String = '' | acc + reg.toString())
    +'\n    }\n'
    ;

helper context PivotVxUML!Region def :toString() : String =
    '\n\n  Region'+self.name+' = new Region("   +self.name +  '",this.stateMachine);'
    +self.vertex->iterate(ver; acc : String = '' | acc + ver.toString(self.name))
    +self.transition->iterate(trans; acc : String = '' | acc + trans.toString(self.name))
    +'\n    this.stateMachine.addRegion('+self.name+');'
    ;

helper context PivotVxUML!Transition def : toString(s:String) : String=
              '/*Ajout Transiton */ tr = new Transition("'+self.name+'",'+ s
             +'.getVertex('+s+'.trouverRangEtat("'+self.source.name+'") ),'
             +s+'.getVertex('+s+'.trouverRangEtat("'+self.source.name+'") ),'+s+');'
    + if not self.guard.oclIsUndefined() then                  ------ ajout contraint------
          '\n  tr.setConstraint('+self.guard.name+');'
        else ''
        endif
    + if not self.effect.oclIsUndefined() then                 ------ ajout Behavior-------
          '\n       tr.setEffect('+self.effect.name    +');'
        else ''
        endif
    + if self.trigger->notEmpty()  then                                    ------ ajout trigger-----------
          if not self.trigger->first().event.oclIsUndefined() then                    -
              if not self.trigger->first().event.signal.oclIsUndefined() then
              '\n                  sig = new '+  self.trigger->first().event.signal.name+ '();'
              +'\n                     signalsList.add(sig);'
              +'\n                     tr.setTrigger(new Trigger(sig));'
              else '\n                        tr.setTrigger(new Trigger());'
              endif
          else '\n                     tr.setTrigger(new Trigger());'
          endif
      else '\n                     tr.setTrigger(new Trigger());'
      endif
    +'\n                  '+s+'.addTransition(tr);'
    ;

helper context PivotVxUML!Vertex def : toString(s:String ) : String=
    '\n     /*  Ajout Etat    */ '+s+'.addVertex( new StateV("'+self.name+'", '+s+') );'
    ;
```

Figure VI.14. Extrait de la bibliothèque Lib_PivotVxUML2Java_Code

VI.3.2.5. Simulateur

L'éditeur de code associé à l'environnement Eclipse permet de visualiser et d'éditer le code généré en utilisant des fonctions externes plus avancées. La Figure VI.15 donne un aperçu du code généré de l'application "Gestion d'une agence de réparation de voitures".

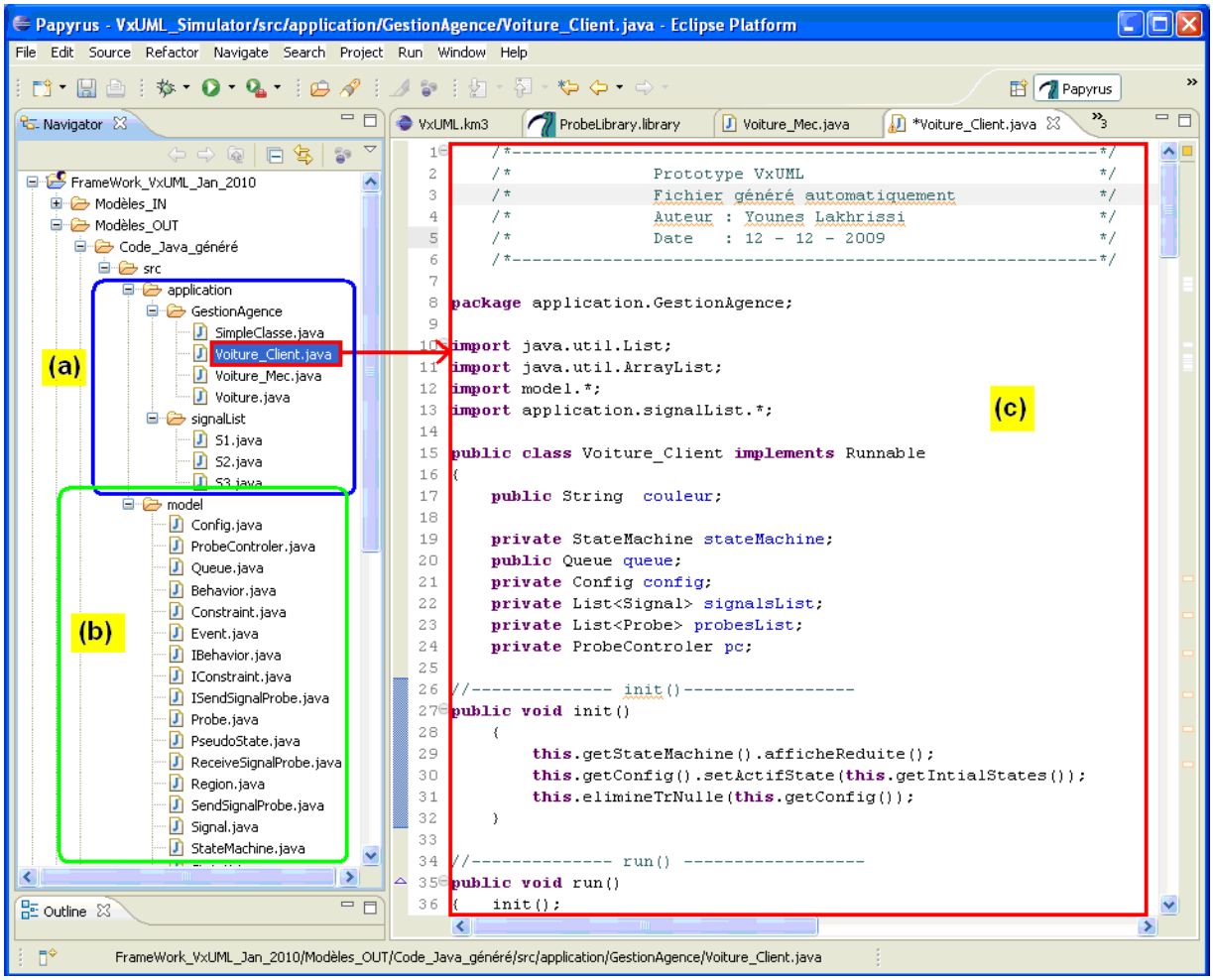


Figure VI.15. Aperçu du code Java généré à partir d'un modèle VxUML

La Figure VI.15-a donne un aperçu des classes métier de l'application (.java). La Figure VI.15-c donne l'exemple du code de la classe Voiture_Client. Nous donnons dans l'annexe C le code Java généré de cette classe. La Figure VI.15-b donne un aperçu des classes générées pour l'application depuis la bibliothèque *SimulationLibrary* afin d'effectuer la simulation. Cette bibliothèque comprend des classes d'aide à la simulation de l'application telles que la classe *ProbeControler.java*, *Config.java*, *Queue.java* et *SMControler.java*. Le rôle de la simulation et d'appliquer des scénarios élémentaires adaptés aux types de sondes utilisées. Pour obtenir un code exécutable, il faut bien sûr compléter le code généré par le code métier implémentant le comportement des méthodes. Nous ne manipulons actuellement que les deux types de sondes *SignalSendProbe* et *SignalReceiveProbe*. Cette partie sur la simulation et l'extension de la bibliothèque par des classes d'aide aux autres types de sondes constitue une des perspectives de ce travail de thèse. La Figure VI.16 donne un aperçu des méthodes de la classe *SMControler.java* utilisée pour la simulation.

```
public boolean evaluerTransition(Transition tr, Event ev)
    {
        return (  evaluerTrigger(tr, ev)  &&  evaluerGuard(tr)  );
    }

public boolean evaluerTrigger(Transition tr, Event ev)
    {
        if( ev ==null )
            {   if( tr.getTrigger()==null || tr.getTrigger().getEvent()==null)
                    return true;
                else return false;
            }
        else if( tr.getTrigger()==null || tr.getTrigger().getEvent()==null)
            return true;
        else
            {   Signal sig = (Signal)ev;
                return tr.getTrigger().getEvent().getClass().getName() == sig.getClass().getName();
            }
    }

public boolean evaluerGuard(Transition tr)
    {
        return tr.getConstraint().eval(this);
    }

public void elimineTrNulle(Config con)
    {
        for(int i=0; i<con.actifStateCount(); i++)
        {
            int taille=con.getActifState().get(i).outgoingTransitionCount();
            int j=0;
            while( j < taille )
            {
                if(con.getActifState().get(i).getOutgoingTransition(j).getTrigger()==null
                    && con.getActifState().get(i).getOutgoingTransition(j).getConstraint()==null
                    &&     con.getActifState().get(i).getOutgoingTransition(j).getTarget() != null)
                    {   con.getActifState().set(i,con.getActifState().get(i).getOutgoingTransition(j).getTarget());
                        taille=con.getActifState().get(i).outgoingTransitionCount();
                        j=0;
                    }
                else j++;
            }
        }
    }
```

Figure VI.16. Aperçu des méthodes de la classe SMControler.java

VI.4. Conclusion

Dans ce chapitre, nous avons décrit le prototype support à VxUML. Nous avons dans un premier temps décrit l'architecture générale et le processus d'implantation du profil VxUML. Puis nous avons présenté les outils support à ce profil, à savoir, un vérificateur de sémantique, un transformateur de modèles VxUML en modèles intermédiaires, un générateur de code Java et un simulateur du code Java généré.

La vérification des modèles VxUML supporte la sémantique de VUML décrite dans [Anwar, 09] ainsi que la sémantique des concepts introduits dans cette thèse. L'opération de génération de code objet à partir d'un modèle VxUML est guidée par des règles de transformation qui respectent la sémantique dynamique de VxUML. Pour l'instant, le langage cible est Java mais les techniques employées sont réutilisables pour cibler d'autres langages objet.

Cette réalisation a profité des avantages offerts par l'approche IDM, notamment en termes d'automatisation de certaines tâches de développement, comme par exemple la transformation de modèles PivotVxUML indépendants d'une plate-forme (PIM) en modèles spécifiques à la plate-forme Java (PSM), et des modèles PSM vers le code. De plus, l'environnement Eclipse nous a permis de réduire considérablement le temps nécessaire pour développer le prototype support de notre approche VxUML. En effet, la plupart des outils utilisés sont supportés par l'environnement Eclipse. Par exemple, pour la transformation de modèles, nous avons utilisé le langage ATL, ainsi que le framework UML2 pour la manipulation des modèles conformes au méta-modèle UML 2.0.

Pour avoir une idée de notre prototype en termes quantitatifs par rapport au nombre de règles, de classes et de lignes de code :

- ProbeLibrary : 20 classes de sondes prédéfinies ;
- Profils utilisés : deux profils, 8 stéréotypes manipulés ;
- Vérificateur sémantique : 15 règles ATL de vérification et 22 helpers (environ 400 lignes de code ATL) ;
- Transformateur PivotVxUML : 16 règles de transformation et 13 helpers (environ 350 lignes de code ATL) ;
- Générateur de code Java et les deux bibliothèques associées : une requête et 39 helpers (environ 1600 lignes de code ATL) ;
- SimulationLibrary : 20 classes (environ 1200 lignes de code Java).

Le prototype est opérationnel, mais néanmoins reste dans une version précoce nécessitant un travail d'extension. Par rapport aux limites de notre prototype, nous pouvons citer : (1) la composition de sondes n'est pas encore supportée par le prototype. En fait, au niveau de la spécification du comportement utilisant les profils développés, on peut déclarer des sondes composées, par contre, nous n'avons pas encore implémenté leur traduction dans le modèle PivotVxUML. (2) Concernant le simulateur, uniquement deux types de sondes sont supportés : SignalSendProbe et SignalReceiveProbe.

CHAPITRE VII. CONCLUSION GENERALE ET PERSPECTIVES

VII.1. Rappel du contexte général et de l'objectif de la thèse

La modélisation par points de vue constitue la thématique générale de notre travail de thèse. C'est une approche de modélisation orientée objet, visant l'analyse et la conception des systèmes complexes avec une démarche centrée acteur. Elle consiste en l'élaboration d'un modèle unique accessible suivant plusieurs points de vue. Le profil UML au sein duquel s'inscrit ce travail supporte la modélisation multivue en permettant l'analyse/conception des systèmes complexes à travers la séparation des points de vue des acteurs du système [Nassar, 05]. Cette séparation facilite la production et la modification éventuelle de modèles spécifiques aux acteurs. Sur le plan méthodologique, VUML propose une démarche centrée utilisateur qui permet d'intégrer la notion de point de vue dans le processus de développement. Selon cette démarche, plusieurs modèles de conception sont développés par différents concepteurs. Ces modèles doivent être ensuite composés pour produire un modèle VUML unique représentant la conception globale du système.

Si le principe d'une décomposition d'un système présente indéniablement des avantages, notamment la diminution de la complexité du développement, par contre composer les modèles partiels résultant de la décomposition initiale est un processus délicat. Le travail réalisé dans le cadre de la thèse d'Anwar [Anwar, 09] a été focalisé sur la proposition d'une approche générique de composition des modèles structurels issus de conceptions séparées.

Cependant, les travaux réalisés sur le profil VUML par A. Anwar ne couvrent pas les aspects comportementaux de la modélisation. A. Anwar a traité les aspects structuraux liés au partage de données et à la composition des vues sans traiter la manière dont ces vues se comportent, ni comment les composer afin de représenter le comportement global des objets multivue (instances de classes multivue).

Rappel de l'objectif de la thèse

Les travaux effectués dans le cadre de cette thèse ont eu pour objectif de combler le manque identifié ci-avant en dotant le profil VUML de nouveaux mécanismes permettant d'exprimer les besoins comportementaux d'un système. Les questions principales abordées dans cette thèse ont été les suivantes :

- comment spécifier les comportements individuels des vues de manière à respecter la démarche de développement de VUML, et comment représenter les interactions entre les vues ?

- comment composer les comportements des vues pour représenter le comportement global des objets multivue, et comment garantir la cohérence et l'intégrité du comportement composé ?

VII.2. Apport du présent travail

Pour décrire le comportement des vues dans le cadre d'un objet multivue, deux possibilités s'offraient à nous. La première consistait à réutiliser tels quels les mécanismes de spécification du comportement et des communications entre objets de UML. La deuxième consistait à trouver de nouveaux mécanismes adaptés, qui étendent UML. Nous avons exploré dans un premier temps la première approche et compte tenu de ses inconvénients, avons opté pour la deuxième, comme cela est résumé ci-dessous.

Approche fondée sur les concepts d'UML

Cette approche propose une méthode de description séparée et de coordination des machines à états associées aux objets-vue et à l'objet-base. Le résultat essentiel est l'élaboration, pour une classe multivue donnée, de deux types de machines à états particulières attachées soit à une classe « base », soit à une classe « vue ». Ces machines à états, qui suivent la spécification UML2.0 [OMG-UML], ont pour rôle de capturer le comportement dynamique des instances de la classe multivue considérée [Lakhrissi et al., 07], [Lakhrissi et al., 08a]. Une machine-vue représente le cycle de vie d'un objet-vue, tandis qu'une machine-base a pour but de créer la cohérence et la coordination entre les machines-vue, et de spécifier le comportement commun aux acteurs. Toutefois, l'expérience avec cette approche montre que l'intégration de machines-vue développées séparément peut requérir de nombreuses modifications et de lourdes adaptations de ces machines à états au moment de la fusion, dès lors qu'il y a une interdépendance forte entre les vues. Or, dans un contexte de passage à l'échelle caractérisé par des systèmes complexes ayant de nombreux points de vue, cette inter-dépendance est inévitable.

Approche par extension des concepts d'UML

Nous avons donc opté pour une autre approche, générique dans son principe et ajoutant dans VUML des mécanismes nouveaux, spécifiques de modélisation et de composition du comportement des objets-vue [Ober et al., 08b]. Nous avons introduit pour cela la notion *de sonde d'événements* pour spécifier des communications implicites entre les objets-vue à travers des observations d'événements. Ceci permet de découpler des spécifications qui sont a priori fortement interconnectées, de les concevoir séparément, puis de les intégrer sans avoir à les modifier. La démarche que nous avons déployée dans la thèse à cet effet est la suivante :

- ***Définition de la notion de sonde d'événements.*** Nous avons défini le concept de sonde d'événements qui est basé sur l'observation des événements du système en exécution. D'abord nous avons identifié les différents types de sondes et les paramètres adéquats qui les caractérisent. Ensuite nous avons défini un ensemble de concepts permettant d'enrichir la manipulation des différents types de sondes, tels que la projection d'une sonde, la dérivation d'une sonde et la composition de sondes.

- ***Proposition d'une représentation compatible avec UML.*** Nous avons adapté notre proposition aux préconisations d'UML. Cela a été réalisé en proposant des extensions des concepts UML, tels que la définition d'un nouveau type de *trigger* pour les transitions basées sur l'observation d'événements, la définition d'un élément de référencement utilisé par les classes utilisatrices pour référencer les sondes, etc.

- ***Définition du profil VUML étendu.*** Nous avons défini le profil Probe_profile capturant les concepts dédiés au traitement des aspects comportementaux dans VUML. En plus de la définition des éléments du profil (stéréotypes, *tagged values*, classes de librairie prédéfinies), nous avons développé des règles de bonne formation de la sémantique statique (well formedness rules) en utilisant OCL.

- ***Raffinement de la démarche associée à VUML.*** Nous avons complété la démarche associée à VUML (qui ne traitait que les aspects structuraux) pour pouvoir utiliser d'une manière méthodique les nouveaux mécanismes dédiés au traitement du comportement. Désormais, le processus de développement VUML offre une modélisation par points de vue complète, c'est-à-dire traitant à la fois les aspects structurel et comportemental.

- ***Réalisation d'une étude de cas détaillée.*** Afin de montrer l'intérêt des concepts introduits et de valider notre approche, nous avons développé une étude de cas significative traitant la modélisation multivue d'un système de gestion d'une agence de réparation de voitures. Nous avons réalisé à cet effet une modélisation par points de vue traitant les deux aspects, structurel et comportemental.

- ***Réalisation d'un prototype.*** Pour valider concrètement notre approche, nous avons développé, selon une approche IDM, un générateur de code pour prototypage rapide, qui prend en entrée une spécification de système développée dans le profil VxUML et produit en sortie une version exécutable du système dans un langage orienté objet. Ce générateur utilise les techniques de transformation de modèles liées à la génération de code, et notamment les transformations de modèles indépendants d'une plate-forme (PIM) vers des modèles spécifiques à une plate-forme (PSM), et des modèles PSM vers le code ; il a été développé, dans un premier temps, avec Java comme langage cible.

L'utilisation des sondes dans VxUML apporte des solutions aux deux problèmes liés à la spécification et à la composition du comportement évoqués ci-dessus. Premièrement, elle permet de définir le comportement des vues indépendamment les unes des autres dans la phase de conception décentralisée. Cela est réalisé sans spécifier de mécanisme de communication explicite pour faire passer l'information entre les différents modèles-vue. En effet, le concepteur se décharge, en déclarant des sondes abstraites, de la manière dont le comportement recherché va être exprimé, car cette tâche est déléguée aux sondes déclarées. Deuxièmement, l'utilisation des sondes offre un principe simple pour la composition des comportements dans la phase de fusion, et évite de devoir modifier les modèles-vue. En fait, au lieu de modifier les modèles-vue pour réaliser la composition, on procède à une synchronisation de ces derniers en agissant sur les sondes déclarées. Cette mise en cohérence se fait à travers la concrétisation de la définition des sondes abstraites utilisées dans les différents modèles-vue.

VII.3. Perspectives

Le travail réalisé sur le profil VUML et plus précisément sur l'aspect comportemental, ouvre de nombreuses perspectives, tant sur le plan de l'outillage que sur le plan théorique.

- Perspectives au niveau de l'outillage

Un certain nombre de travaux sont en cours de réalisation. Nous pouvons citer en premier lieu le travail sur le prototype support au profil VxUML que nous sommes en train de finaliser sous forme d'un plug-in Eclipse. Comme suite à ce travail sur le prototype, nous pouvons énoncer les perspectives suivantes :

- Développer un outil de validation des modèles partiels par points de vue

Dans la phase de développement décentralisée, il est possible de développer un outil de validation des modèles-vue élaborés séparément. En effet, un modèle-vue est déjà un modèle complet qui peut, avec un outillage adapté permettant de simuler l'activation des sondes, être exécuté dans le but d'être validé.

- Compléter le développement de l'outil de simulation

La simulation d'un système nécessite l'injection de scénarios de test. Nous comptons développer une bibliothèque de classes prédéfinies pour réaliser des tests. Cette bibliothèque sera basée et adaptée pour le test des différents types de sondes.

- Compléter le développement du générateur de code

Le générateur de code actuel cible le langage Java, nous souhaitons l'enrichir afin qu'il puisse cibler d'autres langages à objet tels que C++, Ruby, etc.

- Appliquer notre démarche sur d'autres cas d'étude

Nous avons appliqué notre prototype sur divers exemples permettant de valider nos choix. Ces exemples sont extraits du cas d'étude "Gestion d'une agence de réparation de voitures" dont la taille et la complexité sont significatives. Néanmoins, pour un passage à l'échelle, la mise en œuvre avec notre outil d'un développement logiciel de taille industrielle permettra une véritable validation des résultats de cette thèse.

- Intégrer les outils réalisés sur la composition de modèles structurels

Les travaux réalisés dans le cadre de la thèse d'A. Anwar [Anwar, 09] ont donné lieu au développement du prototype VMT. Nous souhaitons intégrer l'ensemble des outils développés dans le cadre du projet VUML dans un prototype unique.

- Perspectives au niveau théorique

Suite à nos travaux de recherche sur le profil VUML et sur le concept de sonde d'événements, nous avons identifié plusieurs perspectives, notamment :

- Le tissage des aspects par le concept de sonde d'événements

L'approche par aspect (Aspect-Oriented Programming) décompose le système en unités fonctionnelles de base (ou métiers) d'une application et en fonctionnalités transversales extrinsèques aux exigences métiers. Elle a à traiter la même problématique de composition de modèles que l'approche par points de vue. Nous comptons expérimenter la technique de sonde d'événements, développée dans ce travail de thèse, comme moyen de tissage d'aspects.

- Etude des modèles de description du comportement inter-objets dans VUML

Notre étude a porté sur la modélisation du comportement intra-objet à travers des machines à états. Afin de produire une conception complète prenant en compte les différentes facettes de l'approche par points de vue, il faut compléter ce travail de thèse par le traitement du comportement inter-objets. Cela permettra d'enrichir la démarche VUML dans ses phases amont, notamment dans la phase d'analyse globale où l'on pourra donner une vision plus précise sur les points de vue à considérer dans le système ainsi que leurs collaborations et dépendances partielles dans l'expression des exigences.

- Les patrons de conception multivue

Un de nos objectifs est de proposer des composants génériques ou « patrons multivue » afin de permettre la réutilisation, et donc l'amélioration de la productivité de l'analyse/conception des systèmes complexes. Les points de vue à considérer seront des points de vue « utilisateur final » propres à chaque domaine d'application, mais aussi des points de vue prédéfinis représentant les développeurs (analyste, concepteur, programmeur, testeur, maintenicien, etc.). Les principaux axes envisagés sont : (i) Définir la notion de composant multivue générique sous forme de patrons logiciels multivue ; (ii) Adapter les patrons de conception actuels (mono-vue) pour supporter le développement multivue ; (iii) Elaborer des principes pour la composition de ces patrons ; (iv) Formaliser une sémantique pour ces patrons.

VII.4. Liste des publications

Cette section présente par catégorie, les publications que nous avons réalisées pendant cette thèse.

- **Revues nationales**

  - Mohammed Dahchour, Hamza Rayd, Younes Lakhrissi, Abdelaziz Kriouile. Extension d'UML par les rôles (version étendue de MCSEAI 2006). Dans : la revue électronique des technologies de l'information, Ecole Mohammedia d'Ingénieurs, Rabat - Maroc, Vol. 4, juin 2007.

- **Conférences internationales**

  - Iulian Ober, Bernard Coulette, Younes Lakhrissi. Behavioral modeling and composition of object slices using event observation. Dans ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2008), Toulouse, 28/09/08-03/10/08, Springer, LNCS 5301, p. 219-233, septembre 2008.

- Iulian Ober, Younes Lakhrissi. Observation-based interaction and concurrent aspect-oriented programming. International Conference on Software Engineering Research, Management and Applications (SERA 2008), Prague, Rép. Tcheque, 20/08/2008-22/08/2008, Walter Dosch, Roger Lee (Eds.), Springer, SCI, août 2008.

- **Conférences et workshops nationaux**

  - Younes Lakhrissi, Adil Anwar, Bernard Coulette, Sophie Ebersold, Iulian Ober, Mahmoud Nassar, Abdelaziz Kriouile, VUML, approche de modélisation centrée points de vue, Poster à CAL'09, Nancy, mars 2009.
  - Younes Lakhrissi, Adil Anwar, Bernard Coulette, Sophie Ebersold, Iulian Ober, Composition de modèles par points de vue, Poster à GDR-GPL09, ENSEEIHT, Toulouse, 28-30 janvier 2009.
  - Younes Lakhrissi, Iulian Ober, Bernard Coulette, Mahmoud Nassar, Abdelaziz Kriouile. Prise en compte des aspects comportementaux dans la démarche de modélisation de VUML. Dans : ERTSI, Fontainebleau, 27/05/2008-27/05/2008, Hermès, mai 2008.
  - Younes Lakhrissi, Adil Anwar, Mahmoud Nassar, Abdelaziz Kriouile. Composition des machines à états par points de vue dans VUML. JIMD'2008. 03/07/2008-05/07/2008. ENSIAS Rabat, juillet 2008.
  - Adil Anwar, Younes Lakhrissi, Mahmoud Nassar, Abdelaziz Kriouile. Composition de modèles structurels dans l'approche VUML. JIMD'2008. 03/07/2008-05/07/2008. ENSIAS Rabat, juillet 2008.
  - Younes Lakhrissi, Iulian Ober, Bernard Coulette, Mahmoud Nassar, Abdelaziz Kriouile. Vers la notion de machine à états multivue dans le profil VUML. Dans : WOTIC'07 05/07/2007-06/07/2007, Rabat, juillet 2007.
  - M. Dahchour, H Rayd, Younes Lakhrissi, Abdelaziz Kriouile. Extension d'UML par les rôles. Dans : MCSEAI 2006 (Maghrebian Conference on Software Engineering and Artificial Intelligence), Agadir, 07/12/2006-09/12/2006, Université Ibn Zohr, décembre 2006.
  - Bernard Coulette, Younes Lakhrissi, Mahmoud Nassar, Abdelaziz Kriouile. Notion de patrons multivue - Application au profil VUML. Dans : Workshop OCM-SI, associé à INFORSID 2006, Hammamet, 12/05/2006-14/05/2006, Hermès, mai 2006.

- **Rapports de recherche**

  - Younes Lakhrissi, Bernard Coulette, Iulian Ober, Mahmoud Nassar, Abdelaziz Kriouile. Démarche VUML statique et dynamique - Application à une étude de cas. Rapport de recherche, IRIT/RR-2008-1-FR, IRIT, février 2008, accès: ftp://ftp.irit.fr/IRIT/MACAO/RapportIRITLakhrissi-etal.pdf.
  - Younes Lakhrissi, Rayd Hamza, Extension du langage UML par l'association Role-of, Mémoire du DESA ITM, Faculté des sciences de Rabat, octobre 2004.

BIBLIOGRAPHIE

[Abiteboul et al., 91] S. Abiteboul, A. Bonner. "Objects, Views". Proceedings of ACM SIGMOD, pp 238-247, mai 1991.

[Amyot et al., 03] D. Amyot, A. Eberlein. "An evaluation of scenario notations, construction approaches for telecommunication systems development". Telecommunication Systems, 24(1):61–94, 2003.

[Anwar et al., 07] A. Anwar, S. Ebersold, B. Coulette, M. Nassar, A. Kriouile. "Vers une approche à base de règles pour la composition de modèles. Application au profil VUML". Dans : L'Objet, Hermès Science Publications, Numéro spécial Ingénierie Dirigée par les Modèles, Vol. 13, N. 4/2007, p. 73-103, décembre 2007.

[Anwar et al., 08a] A. Anwar, S. Ebersold, M. Nassar, B. Coulette, A. Kriouile. "A QVT- Based Approach for Model Composition: Application to the VUML Profile". In Proc of ICEIS 2008 Barcelone, pp 360-367, INSTICC Press, 2008.

[Anwar et al., 08b] A. Anwar, S. Ebersold, B. Coulette, M. Nassar, A. Kriouile. "Towrds a generic approach for model composition". Dans : International Conference on Software Engineering Advances, Sliema (Malte), 26/10/08-31/10/08, IEEE Computer Society, p. 84-90, octobre 2008.

[Anwar et al., 08c] A. Anwar, Y. Lakhrissi, M. Nassar, A. Kriouile. "Composition de modèles structurels dans l'approche VUML". Workshop JIMD'2008. 03/07/2008-05/07/2008. ENSIAS Rabat, juillet 2008.

[Anwar, 09] A. Anwar. "Formalisation par une approche IDM de la composition de modèles dans le profil VUML". Thèse de doctorat, Université de Toulouse, décembre 2009.

[Anwar et al., 10] A. Anwar, S. Ebersold, M. Nassar, B. Coulette, A. Kriouile. "A Rule-Driven Approach for composition of Viewpoint-oriented Models". Dans : JOT (Journal of Object Technology). Mars 2010.

[ATL 2005] "The ATL UML to JAVA transformation". Available at http://www.eclipse.org/gmt/atl/atlTransformations/

[ATL, 07] Eclipse/M2M Project Web Page. http://www.eclipse.org/m2m/, 2007.

[ATL-Java] The ATL UML to JAVA transformation. Available at http://www.eclipse.org/gmt/atl/atlTransformations/

[Baniassad et al., 04] E. Baniassad, S. Clarke. "Theme: An approach for aspect-oriented analysis, design". Proc. of the International Conference on Software Engineering, p. 158-167, 2004.

[Bardou, 98a] D. Bardou. "Etude de langages à prototypes, du mécanisme de délégation et de son rapport à la notion de point de vues". Thèse de doctorat en Informatique, LIRMM, université de Montpellier 2, 1998.

[Bardou, 98b] D. Bardou. "Roles, Subjects, Aspects: How do they relate?". Position paper at the Aspect Oriented Programming Workshop. 12th European Conference on Object-Oriented Programming (ECOOP '98), LNCS, vol. 1543, Springer, 1998.

[Barra et al., 04] E. Barra, G. Genova, J. Llorens. "An approach to Aspect Modelling with UML 2.0". In Aspect-Oriented Modeling Workshop, AOM 2004, Lisbon, Portugal, October 2004.

[Basch et al., 03] M. Basch, A. Sanchez. "Incorporating Aspects into the UML". Third International Workshop on Aspect-Oriented Modeling (AOM'03), March 2003.

[Beek, 94] M. Von Der Beeck. "A comparison of Statecharts variants". In FTRTFT '94, volume 863 of LNCS, pages 128-148. Springer-Verlag, 1994.

[Bézivin et al., 05] J. Bézivin, F. Jouault. "Using ATL for Checking Models". In proc. of the International Workshop on Graph, Model Transformation (GraMoT), Tallinn, Estonia, 2005.

[Bézivin et al., 05b] J. Bézivin, F. Jouault, D. Touzet. "An introduction to the ATLAS Model Management Architecture". Rapport de recherche N° 05.01. LINA, université de Nantes, Février 2005.

[Bézivin, 04] J. Bézivin, F. Jouault, P. Rosenthal, P. Valduriez. "The AMMA platform support for modeling in the large, modelling in the small". Research Report LINA, (04.09), 2004.

[Bontemps et al., 04a] Y. Bontemps, P. Heymans. "As fast as sound (lightweight formal scenario synthesis, verification)". In 3rd International Workshop on Scenarios, State Machines: Models, Algorithms,, Tools (SCESM '04), Edinburgh, UK, 2004.

[Bontemps et al., 04b] Y. Bontemps, P.Y. Schobbens, C. Löding. "Synthesis of Open Reactive Systems from Scenario-Based Specifications,". Fundamenta Informaticae, vol. 62, no. 2, pp. 139-169, July 2004.

[Bontemps, 01] Y. Bontemps. "Automated Verification of State-based Specifications Against Scenarios (A Step towards Relating Inter-Object to Intra-Object Specifications)". Master's thesis, University of Namur, rue Grandgagnage, 21 - 5000 Namur(Belgium), June 2001.

[Brill et al., 04] M. Brill, W. Damm, J. Klose, B. Westphal, H. Wittke. "Live Sequence Charts, An Introduction to Lines, Arrows,, Strange Boxes in the Context of Formal Verification". LNCS 3147, pp. 374–399, Springer-Verlag Berlin Heidelberg 2004.

[Budinsky et al., 03] F. Budinsky, D. Steinberg, R. Ellersick. "Eclipse Modeling Framework : A Developer's Guide". Addison-Wesley.

[Buh, 98] R. J. A. Buhr. "Use Case Maps as Architectural Entities for Complex Systems". In: IEEE Transactions on Software Engineering, 24(12), 1131-1155, Dec 1998.

[Caron et al., 03] O. Caron, B. Carré, A. Muller, G. Vanwormhoudt. "A Framework for Supporting Views in Component Oriented Information Systems". OOIS, vol. 2817 de Lecture Notes in Computer Science, Springer, p. 164-178, septembre 2003.

[Carré et al., 90-a] B. Carré, L. Dekker, J.M. Geib. "Multiple, Evolutive Représentation in the ROME language". Actes de TOOLS'90, pp. 101-109, 1990.

[Carré et al., 90-b] B. Carré, J.M. Geib. "The Point of View Notion for Multiple Inheritance". Proceedings of ECOOP/OOPSLA'90, pp. 312-321, 1990.

[Carré, 89] B. Carré. "Méthodologie orientée objet pour la représentation des connaissances, concepts de points de vue, de représentation multiple et évolutive d'objets". Thèse du LIFL, 1989.

[Castejon, 05] H-N. Castejon. "Synthesizing state-machine behaviour from UML collaborations and Use Case Maps". In: Proc. of the 12th Int. SDL Forum, Norway, LNCS 3530, Springer, 2005.

[Charrel et al., 93] P.J. Charrel, D. Galaretta, C. Hanachi, B. Rothenburger. "Multiple Viewpoints for Development of Complex Software". Actes de IEEE International Conference on Systems, Man, Cybernetics, pp. 556-561, 17-20 octobre 1993.

[Clarke, 01] S. Clarke. "Composition of Object-Oriented Software Design Models". PhD thesis, Dublin City University, 2001.

[Clarke, 02] S. Clarke. "Extending Standard UML with Model Composition Semantics". Science of Computer Programming, 44, p. 71-100, 2002.

[Coady et al., 01] Y. Coady, G. Kiczales, M. Feeley, G. Smolyn. "Using Aspect C to Improve the Modularity of Path-Specific Customization in Operating System Code". 8 th European Software Engineering Conference (ESEC), 9 th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), p. 88-98,Vienna, Austria, 2001.

[Cottenier et al., 07] T. Cottenier, A. van den Berg, T. Elrad. "Motorola WEAVR: Model Weaving in a Large Industrial Context". Aspect-Oriented Software Development (AOSD), Vancouver, Canada, 2007.

[Coulette et al., 06] B. Coulette, Y. Lakhrissi, M. Nassar, A. Kriouile. "Notion de patrons multivue - Application au profil VUML". Dans : Workshop OCM-SI, associé à INFORSID 2006, Hammamet, 12/05/2006-14/05/2006, Hermès, mai 2006.

[Coulette et al., 96] B. Coulette, A. Kriouile, S. Marcaillou. "L'approche par points de vue dans le développement orienté objet des systèmes complexes". Revue l'Objet, vol. 2, n°4, pp. 13-20, février 1996.

[Crégut et al., 05] X. Crégut, S. Marcaillou, M. Nassar, B. Coulette. "Un patron de génération de code pour le profil VUML". LMO-OCM'2005, pp. 5-11, Berne, Suisse, mars 2005.

[Cueignet et al., 92] X. Cueignet, V. Lextrait. "Génération de serveur de vues". Thèse de l'université de Sophia Antipolis, décembre 1992.

[Dahchour et al., 04] M. Dahchour, A. Pirotte, E. Zimányi, "A role model, its metaclass implementation". Information Systems Journal, volume 29, p. 235-270, Elsevier, 2004.

[Dahchour et al., 06] M. Dahchour, H. Rayd, Y. Lakhrissi, A. Kriouile, "Extension d'UML par les rôles". Proc. of the 9th Maghrebian Conference on Information Technologies (MCSEAI 2006), Agadir, Morocco, December 2006.

[Dahchour et al., 07] M. Dahchour, H. Rayd, Y. Lakhrissi, A. Kriouile. "Extension d'UML par les rôles" (version étendue de MCSEAI 2006). Dans : la revue électronique des technologies de l'information ETI, Ecole Mohammadia d'Ingénieurs, Rabat - Maroc, Vol. 4, juin 2007.

[Damm et al., 01] W. Damm, D. Harel. "LSCs: Breathing life into message sequence charts". Formal Methods in System Design, 19(1):45–80. Preliminary version appeared in Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99), 2001.

[Debrauwer 98] L. Debrauwer. "Des vues aux contextes pour la structuration fonctionnelle de bases de données à objets en CROME". Thèse de doctorat en Informatique, LIFL, Université des Sciences et Technologies de Lille, décembre 1998.

[Dekker, 94] L. Dekker. "FROME : Représentation multiple et classification d'objets avec points de vue". Thèse de l'Université de Lille, juin 1994.

[Del Fabro et al., 06] M. Didonet Del Fabro, J. Bézivin, P. Valduriez. "Weaving Models with the Eclipse AMW plugin". In: Eclipse Modeling Symposium, Eclipse Summit Europe 2006, Esslingen, Germany, 2006.

[Eclipse, 07] QVT Operational - M2M component. http://www.eclipse.org/m2m/qvto/doc/M2M-QVTO.pdf

[El Asri et al., 04] B. El Asri, M. Nassar, A. Kriouile, B. Coulette. "Views, subjects, roles, aspects : A comparison along software lifecycle". Proceedings of 6th International Conference on Enterprise Information Systems ICEIS'04, Porto-Portugal, April 2004.

[El Asri et al., 05] B. El Asri, M. Nassar, B. Coulette, A. Kriouile. "MultiViews component for information development". Proceedings of the 7th International Conference on Enterprise Information Systems (ICEIS'2005), pp. 217-225, Miami, USA, May 24-28, 2005.

[El Asri, 05] B. El Asri. "Vers des composants multivues distribués". Thèse nationale, l'ENSIAS de Rabat, octobre 2005.

[Epsilon, 06] Epsilon SubProject 2006. http://www.eclipse.org/gmt/epsilon/

[Finkelsetin et al., 92] A. Finkelsetin, J. Kramer, B. Nuseibeh, L. Finkelstein, M. Goedicke. "Viewpoints: A Framework for Integrating Multiple Perspectives in System Development". International Journal of Software Engineering, Knowledge Engineering 2(1):31-58, World Scientific Publishing Co, March 1992.

[Finkelstein et al., 90] A. Finkelstein, J. Kramer, M. Goedicke. "Viewpoint Oriented Software Development". Proceedings of Software Engineering, Applications Conference, p. 337-351, Toulouse, December 1990.

[France et al., 04a] R. France, D. Kim, S. Ghosh, E. Song. "A UML-based pattern specification technique", IEEE Trans. Sofw. Eng., 2004.

[France et al., 07] R. France, F. Fleurey, R. Reddy, B. Baudry, S. Ghosh. "Providing Support for Model Composition in Metamodels". In procc of the 11th IEEE EDOC conference, pp 253-264. 2007.

[GEF, 07] The Graphical Editing Framework (GEF). http://www.eclipse.org/

[GMF, 07] The Graphical Modeling Framework (GMF). http://www.eclipse.org/

[Gottlob et al., 96] G. Gottlob, M. Schrefl, B. Röck B. "Extending object-oriented systems with roles". ACM Trans. Office Information Systems, 14 (3), p. 268-296, 1996.

[Griffin, 04] C. Griffin. "Transformations in Eclipse". Workshop on Model-Driven Development. WMDD / IBM / © ATHENA Consortium 2004.June 2004.

[Harel et al., 00] D. Harel, H. Kugler. "Synthesizing state-based object systems from LSC specifications". Int. J. of Foundations of Computer Science (IJFCS)., 13(1):5–51, Febuary 2002. (Also,Proc. Fifth Int. Conf. on Implementation, Application of Automata (CIAA 2000), July 2000, Lecture Notes in Computer Science, Springer-Verlag, 2000.).

[Harel et al., 03] D. Harel, H. Kugler, R. Merlly, A. Pnueli. "Smart Play-Out". OOPSLA'03, Aaheim, California, USA. ACM 1-58113-751-6/03/0010, October 2003.

[Harel et al., 04b] D. Harel, H. Kugler. "The RHAPSODY semantics of statecharts (on, on the executable core of the UML)" (preliminary version). In SoftSpez Final Report, LNCS 3147, pages 325–354. Springer, 2004.

[Harel et al., 05a] D. Harel, H. Kugler, A. Pnueli. "Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements". Springer-Verlag Berlin Heidelberg 2005.

[Harel et al., 05b] D. Harel, H. Kugler, G. Weiss. "Some Methodological Observations Resulting from Experience Using LSCs, the Play-In/Play-Out Approach". Proc. Scenarios: Models, Algorithms, Tools, Lecture Notes in Computer Science, Springer-Verlag, 2005 .

[Harel et al., 96] D. Harel, A. Naamad. "The STATEMATE semantics of statecharts". ACM Transactions on Software Engineering, Methodology, 5(4):293–333, 1996.

[Harel et al., 97] D. Harel, E. Gery. "Executable object modeling with statecharts". IEEE Computer , pp. 31-42, July 1997.

[Harel et Marelly, 03] D. Harel, R. Marelly. "Specifying, Executing Behavioral Requirements: The Play In/Play-Out Approach". Software, System Modeling (SoSyM), pp 82-107, 2003.

[Harel, 84] D. Harel. "Statecharts: A Visual Formalism for Complex Systems". Science of Computer Programming 8 (1987), 231–274. (Preliminary version: Technical Report CS84-05, The Weizmann Institute of Science, Rehovot, Israel, February 1984.)

[Harel, 87] D. Harel. "Statecharts: A visual formalism for complex systems". Science of Computer Programming, 8(3):231–274, June 1987.

[Harrison et al., 93] W.Harrison, H. Ossher. "Subject-oriented programming : a critique of pure objects". Proceedings of OOPSLA'93, Washington D.C., pp. 411-428, Se, 1993.

[Huizing, 91] C. Huizing. "Semantics of Reactive Systems: Comparison". PHD thesis, Eindhoven University of Technology, 1991.

[Humberto et al., 05] N. Humberto, M. Castejon. "Synthesizing State-Machine Behaviour from UML Collaborations, Use Case Maps". SDL 2005, LNCS 3530, pp. 339–359, Springer-Verlag Berlin Heidelberg 2005.

[Hyades, 02] Eclipse Hyades Project. http://www.eclipse.org/hyades/

[IFx-site] http://www-if.imag.fr/IFx/

[ITU-MSC, 00] ITU-TS, Recommendation Z.120 (11/99): MSC 2000. ITU-TS, Geneva, 1999.

[ITU-MSC-B, 95] ITU-TS, Recommendation Z .120: Message Sequence Chart (MSC)- Annex B : Algebraic Semantics of Message Sequence Charts . ITU--TS, Geneva, 1995.

[ITU-SDL, 00] ITU-T, Recommendation Z.100: Specification, Description Language (SDL), 2000.

[ITU-UCM, 02] ITU-T, URN Focus Group (2002), Draft Rec. Z.152 - UCM: Use Case Map Notation (UCM). Geneva, 2002.

[ITU-URN] ITU-T, Recommendation Z.150, User Requirements Notation (URN)- Language Requirements, Framework, Geneva, Switzerland. http://www.UseCaseMaps.org/urn/

[Jacobson et al., 04] I. Jacobson, P-W Ng. "Aspect-Oriented Software Development with Use Cases". Addison-Wesley, 2004.

[Java-meta, 05] http://www.eclipse.org/m2m/atl/atlTransformations/UML2Java/ExampleUML2Java%5Bv00.01%5D.pdf

[Jouault et al., 05] F. Jouault, I. Kurtev. "Transforming Models with ATL. In Proceedings of the Model Transformations in Practice". Workshop at Models 2005, Montego Bay, Jamaica 2005.

[Jouault et al., 06a] F. Jouault. "Contribution à l'étude des langages de transformation de modèles". Thèse de doctorat, Université de Nantes, septembre

[Jouault et al., 06c] F. Jouault, I. Kurtev. "On the Architectural Alignment of ATL, QVT". In: Proceedings of the 2006 ACM Symposium on Applied Computing (SAC 06). ACM Press, Dijon, France, pages 1188—1195, 2006.

[Kiczales et al., 01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold. "An Overview of AspectJ". Proceeding of ECOOP'01, Springer Verlag LNCS2072, 2001.

[Kiczales, 97] G. KICZALES. "Aspect-Oriented Programming". European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag LNCS 1241, Finland, June 1997.

[Klein, 06] J. Klein. "Aspects Comportementaux et Tissage". Thèse de l'Université de Rennes 1, Rennes, Décembre 2006.

[Kriouile, 95] A. Kriouile, "VBOOM, une méthode orientée objet d'analyse et de conception par points de vue". thèse d'Etat de l'université Mohammed V de Rabat, 1995.

[Kristensen, 96] B.B. Kristensen. "Object-oriented modeling with roles". Proc. of the Int. Conf. on Object Oriented Information Systems, OOIS'95, Springer, Berlin, p.57-71, Dublin, Ireland 1996.

[Krüger, 00] I. H. Krüger. "Distributed System Design with Message Sequence Charts". PhD thesis, Technischen Universität München, July 2000.

[Lakhrissi et al., 07] Y. Lakhrissi, I. Ober, B. Coulette, M. Nassar, A. Kriouile. "Vers la notion de machine à états multivue dans le profil VUML". Dans : Workshop WOTIC 05/07/2007-06/07/2007Rabat, juillet 2007.

[Lakhrissi et al., 08a] Y. Lakhrissi, B. Coulette, I. Ober, M. Nassar, A. Kriouile. "Démarche VUML statique et dynamique - Application à une étude de cas". Rapport de recherche, IRIT/RR-2008-1-FR, IRIT, février 2008.

[Lakhrissi et al., 08b] Y. Lakhrissi, I. Ober, B. Coulette, M. Nassar, A. Kriouile. "Prise en compte des aspects comportementaux dans la démarche de modélisation de VUML". Dans : ERTSI, associé à la conférence INFORSID, Fontainebleau, 27/05/2008-27/05/2008, Hermès, mai 2008.

[Lakhrissi et al., 08c] Y. Lakhrissi, A. Anwar, M. Nassar, A. Kriouile. "Composition des machines à états par point de vue dans VUML". Workshop JIMD'2008. 03/07/2008-05/07/2008. ENSIAS Rabat, juillet 2008.

[Le Guennec, 01] A. Le Guennec. "Génie Logiciel et Méthodes Formelles avec UML Spécification, Validation et Génération de tests". Thèse de l'Université de Rennes 1, Rennes 2001.

[Le Moigne, 90] J.L. Le Moigne. "La modélisation des systèmes complexes". Dunod, 1990.

[Liang et al., 06] H. Liang, J. Dingel et Z. Diskin. "A Comparative Survey of Scenario-based to State-based Model Synthesis Approaches". SCESM'06, Shanghai, China, May 2006.

[Marcaillou et al., 94] S. Marcaillou, A. Kriouile, B. Coulette. "VBOOL : une extension d'Eiffel intégrant le concept de points de vue". actes de MCSEAI'94, pp. 115-125, Rabat, Avril 1994.

[Marcaillou, 95] S. Marcaillou. "Intégration de la notion de points de vue dans la modélisation par objets – Le langage VBOOL". thèse de l'université Paul Sabatier de Toulouse, 1995.

[Marino, 93] O. Marino. "Raisonnement classificatoire dans une représentation à objets multi-points de vue". thèse de l'Université Joseph Fourier- Grenoble 1, novembre 1993.

[Marzak, 97] A. Marzak. "Conception de VBTOOL, outil support de la méthode VBOOM, réalisation des fonctionnalités : Analyse et conception". Thèse pour l'obtention du diplôme de spécialité de 3ème cycle de l'université Mohamed V, 1997.

[Mili et al., 01] H. Mili, H. Mcheick, J. Dargham, S. Dalloul. "Distribution d'objets avec vues". Revue L'Objet-7/2001, LMO'2001, pp. 27-44, 2001.

[Muller et al., 03] A. Muller, O. Caron, B. Carré, G. Vanwormhoudt. "Réutilisation d'aspects fonctionnels des vues aux composants". Revue RSTI-L'objet, vol. 9, n°1-2, LMO'2003, pp. 241-255, 2003.

[Muller et al., 05] P-A. Muller, F. Fleury, J-M. Jezequel. "Weaving executability into object-oriented meta-languages". In Proceedings of MODELS/UML 2005, pages 264–278, Montego Bay, Jamaica, October 2005.

[Muller, 06] A. Muller. "Construction de systèmes par application de modèles paramétrés". Thèse de l'Universit´e de Lille 1, 2006.

[Nassar et al., 03] M. Nassar, B. Coulette, X. Crégut, S. Marcaillou, A. Kriouile. "Towards a View based Unified Modeling Language". Proceedings of 5th International Conference on Enterprise Information Systems ICEIS'03, pp. 257-265, Angers, April 2003.

[Nassar et al., 04] M. Nassar, B. Coulette, A. Kriouile. "Génération de code dans VUML". Journal Marocain d'Automatique, d'Informatique et de Traitement du Signal, article sélectionné de la conférence COPSTIC'03, 2004.

[Nassar et al., 09] M. Nassar, A. Anwar, S. Ebersold, B. El Asri, B. Coulette, A. Kriouile. "A Code Generation in VUML profile: a Model Driven Approach". 7th IEEE/ACS AICCSA 2009. IEEE Computer Society Press, Rabat, May 10-13, 2009.

[Nassar, 03] M. Nassar. "VUML : a Viewpoint oriented UML Extension". Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'2003 - Doctoral symposium). pp. 373-376, Montreal, Canada, October 6-10, 2003.

[Nassar, 05] M. Nassar. "Analyse/conception par points de vue : le profil VUML". Thèse INPT, Toulouse, 28 septembre 2005.

[Nicolas et al., 05] H. Nicolas, C. Martinez. "Synthesizing State-Machine Behaviour from UML Collaborations, Use Case Maps". In SDL Forum, pages 339–359, 2005.

[Ober et al., 06] I. Ober, S. Graf and I. Ober. "Validating timed UML models by simulation and verification". International Journal of Software Tools for Technology Transfer (STTT), Volume 8, Number 2, pages 128-145, Springer Verlag, April, 2006.

[Ober et al., 08a] I. Ober, Y. Lakhrissi. "Observation-based interaction, concurrent aspect-oriented programming". Dans : International Conference on Software Engineering Research, Management, Applications (SERA 2008), Prague, Rép. Tcheque, 20/08/2008-22/08/2008, Walter Dosch, Roger Lee (Eds.), Springer, SCI, août 2008.

[Ober et al., 08b] I. Ober, B. Coulette, Y. Lakhrissi. "Behavioral modelling, composition of object slices using event observation". Dans MODELS 2008 (ACM/IEEE 11th International Conference on Model Driven Engineering Languages, Systems), Toulouse 28/09/2008-03/10/2008, Springer, 2008.

[ObjectGeode-site] ObjectGeode, available at http://www.telelogic.com/products/objectgeode/.

[Omega-site] http://www-omega.imag.fr/tools/IFx/IFx.php

[OMG-CORBA] OMG, CORBA Components , version 3.0 full specification. OMG document formal/02-06-65. June 2002. http://www.omg.org/cgi-bin/doc?formal/02-06-65

[OMG-EMOF, 06] [OMG, 06] Object Management Group, Inc. Meta Object Facility (MOF) 2.0 Core

[OMG-MOF 08] Object Management Group, Inc. Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Specification, version 1.0, avril 2008.

[OMG-MOF] OMG 2002. OMG/MOF Meta Object Facility (MOF) 1.4. Final Adopted Specification Document. formal/02-04-03.

[OMG-OCL] OMG 2003, UML2 OCL Final Adopted Specification, http://www.omg.org/docs/ptc/03-10-14.pdf.

[OMG-UML] Object Management Group, Inc. Unified Modeling Language (UML) 2.1.2 Superstructure, novembre 2007. http://www.omg.org/uml.

[OMG-XMI, 03d] XML Metadata Interchange (XMI), v2.0. www.omg.org/cgi-bin/doc?formal/2003-05-02

[OMG-XML] XML Metadata Interchange (XMI), v2.0. www.omg.org/cgi-bin/doc?formal/2003-05-02

[Omondo, 01] Omondo Eclipse UML. http://www.eclipsedownload.com/index.html

[Ossher et al., 01] H. Ossher, P. Tarr. "Using multidimensional separation of concerns to (re)shape evolving software". Communications of the ACM, Vol. 44, No. 10, pp. 43-50, October 2001.

[Ossher et al., 95] H. Ossher, M. Kaplan, W. Harrison, A. Katz, V. Kruskal. "Subject-oriented composition rules". Proceedings of the ACM Conference on Object-Oriented Systems, Languages,, Applications, Austin, TX, OOPSLA'1995, pp. 235-250, Oct 1995.

[Peltier, 02] M. Peltier. "Transformation entre un profil UML et un métamodèle MOF". Revue l'Objet, vol. 8, n°1-2/2002, LMO'2002, p. 25-40, 2002.

[Pernici, 90] B. Pernici. "Objects with roles". Proceedings of the ACM--IEEE Conference on Office Information Systems, Cambridge, MA, 1990.

[Pilone et al., 07] D. Pilone, N. Pitman. "UML2 In A Nutshell". Shroff Publishers & Distributors O'REILLY, 2007.

[Pnueli et al., 91] A. Pnueli, M. Shalev. "What is in a step: On the semantics of Statecharts". In TACS '91, volume 526 of LNCS, pages 244--264. Springer-Verlag, 1991.

[PragmaDev-site] PragmaDev: RTDS V3.1, http://www.pragmadev.com/

[Reddy et al., 06] Y. Reddy, S. Ghosh, R. France, G. Straw, J-M. Bieman, N. McEachen, E. Song, G. Georg. "Directives for Composing Aspect-Oriented Design Class Models". Transactions of Aspect-Oriented Software Development, Vol.1, No. 1, LNCS 3880, p75-105, Springer, 2006.

[Rhapsody-Guide] I-Logix. Rhapsody 6.0 User Guide.

[Rhapsody-Tutorial] I-Logix. Tutorial for Rhapsody in J (Release 4.1 MR2), 2003.

[Riehle et al., 98] D. Riehle. "Framework Design: A Role Modeling Approach". PhD thesis, No. 13509. Zrich, Switzerland, ETH Zrich, 2000.

[Rieu et al., 92] D. Rieu, G.T. Nguyen. "Object Views for Engineering Databases". Actes de "third international conference on data, knowledge systems for manufacturing, engineering, AFCET'92", pp. 335-349, Lyon, mars 1992.

[Rumbaugh et al., 96] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy et W. Lorensen. "OMT : modélisation et conception orientées objet". Prentice-Hall, 1996.

[Softeam, 99] Softeam. "Profiles UML et langage J : Contrôlez totalement le développement d'applications avec UML". White Paper, 1999.

[Soley et al., 00] Soley et al., MDA Model Driven Architecture, by Richard Soley and the OMG Staff Strategy Group, Object Management Group White Paper, Draft 3.2 - November 27, 2000.

[Spinczyk et al., 02] O. Spinczyk , G.,reas, S.P. Wolfgang. "AspectC++: an aspect-oriented extension to the C++ programming language". Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile, embedded applications, Sydney, Australia, February 01, 2002.

[Steimann, 01] F. Steimann. "Role = Interface: a merger of concepts". Journal of Object-Oriented Programming, vol. 14(4), pp. 23–32, 2001.

[Straw et al., 04] G. Straw, G. Georg, E. Song, S. Ghosh, R. France, J-M. Bieman. "Model composition directives". In Proceedings of 7th International Conference on The Unified Modeling Language. Model Languages, Applications (UML 2004), volume 3273 of LNCS, pages 84–97. Springer, 2004.

[Sztipanovits et al., 97] J. Sztipanovits, G. Karsai. "Model-Integrated Computing". Computer, Apr. 1997, pp. 110-112.

[Tarr et al., 99] P.L. Tarr, H. Ossher, W. Harrison, M. Stanley, Jr. Sutton. "N Degrees of Separation : Multi-Dimensional Separation of Concerns". International Conference on Software Engineering, pp. 107-119, 1999.

[Tau-site] Telelogic Tau, available at http://www.telelogic.se/products/tau/.

[Uchitel, 03] S. Uchitel. Elaboration of Behaviour Models, Scenario based Specifications using Implied Scenarios. PhD thesis, Imperial College London, January 2003.

[UCM] Use Case Maps Web Page, UCM Users Group, 1999. http://www.UseCaseMaps.org

[Whittle et al., 00] J. Whittle, J. Schumann. "Generating statechart designs from scenarios". In 22nd International Conference on Software Engineering (ICSE '00), pages 314–323, ACM Press, New York, NY, USA, 2000.

[Ziadi et al., 04] T. Ziadi, L. Helouet, J. Jézéquel. "Revisiting statechart synthesis with an algebraic approach". In 26th International Conference on Software Engineering (ICSE '04), pages 242–251, IEEE Computer Society, Washington, DC, USA, 2004.

[Zito et al., 06] A. Zito, Z. Diskin, J. Dingel. "Package Merge in UML 2: Practice vs. Theory?". Proc. of the 9th International Conference on Model Driven Engineering Languages, Systems (MoDELS 2006), Genoa, Italy, October 2006.

ANNEXE A : META-MODELE PIVOTVXUML EN LANGAGE KM3

```
package PIVOTVXUML
{
abstract class PivotVxUMLElement
    {  attribute name : String;
    }

class PivotVxUMLPackage extends PivotVxUMLElement
    {  attribute path : String;
       reference classes[*] container : Class oppositeOf owner;
       reference signals[*] container : Signal oppositeOf owner;
       reference types[*] container : PivotVxUMLType oppositeOf _package;
       reference objects[*] container : Object oppositeOf owner;
    }

class PivotVxUMLType extends PivotVxUMLElement
    {  reference typedField[*] : Field oppositeOf fieldType;
       reference typedargument[*] : Argument oppositeOf argType;
       reference _package : PivotVxUMLPackage oppositeOf types;
    }

class Object extends PivotVxUMLType
    {  reference instanceOf [0-1] : Class oppositeOf instances;
       reference owner : PivotVxUMLPackage oppositeOf objects;
       attribute isProbe : Boolean;
    }

class Class extends PivotVxUMLType
    {  attribute isAbstract : Boolean;
       attribute isFinal : Boolean;
       attribute isPublic : Boolean;
       attribute isStatic : Boolean;
       reference fields[*] ordered container : Field oppositeOf ownerclass;
       reference method[*] container : Method oppositeOf ownerclass;
       reference constructor[1-*] : Method;
       reference extend[0-1] : Class;
       reference owner : PivotVxUMLPackage oppositeOf classes;
       reference ownedBehavior [*] container : StateMachine oppositeOf ownedClass;
       reference body : ClassBody oppositeOf ownedClass;
       attribute isView : Boolean;
       attribute isBase : Boolean;
       attribute isClassProbe : Boolean;
       reference views [*] : Class oppositeOf base;
```

```
        reference base : Class oppositeOf views;
        reference instances [*] : Object oppositeOf instanceOf;
    }

class Event extends PivotVxUMLElement
    {   reference trigger [*] : Trigger oppositeOf event;
        reference signal : Signal oppositeOf event;
    }

class Signal extends PivotVxUMLElement
    {   reference event [*] : Event oppositeOf signal;
        reference fields[*] ordered container : Field oppositeOf ownerSignal;
        reference owner : PivotVxUMLPackage oppositeOf signals;
    }

class Constraint extends PivotVxUMLElement
    {   reference transition : Transition oppositeOf guard;
        attribute body : String;
    }

class Trigger extends PivotVxUMLElement
    {   reference transition : Transition oppositeOf trigger;
        reference event : Event oppositeOf trigger;
    }

class Behavior extends PivotVxUMLElement
    {   reference transition : Transition oppositeOf effect;
        attribute body : String;
    }
class StateMachine extends PivotVxUMLElement
    {   reference ownedClass : Class oppositeOf ownedBehavior;
        reference region[*] container : Region oppositeOf ownedStateMachine;
    }

class Region extends PivotVxUMLElement
    {   reference ownedStateMachine : StateMachine oppositeOf region;
        reference vertex [*] container : Vertex oppositeOf region;
        reference transition [*] container : Transition oppositeOf region;
    }

class Vertex extends PivotVxUMLElement
    {   reference region : Region oppositeOf vertex;
        reference outgoingTransition [*] : Transition oppositeOf source;
        reference incomingTransition [*] : Transition oppositeOf target;
    }

class Transition extends PivotVxUMLElement
    {   reference region : Region oppositeOf transition;
        reference source : Vertex oppositeOf outgoingTransition;
        reference target : Vertex oppositeOf incomingTransition;
        reference guard [0-1] container : Constraint oppositeOf transition;
        reference trigger [*] container : Trigger oppositeOf transition;
        reference effect [0-1] container : Behavior oppositeOf transition;
    }
```

```
class Field extends PivotVxUMLElement
    {   attribute isFinal : Boolean;
        attribute isStatic : Boolean;
        attribute isPublic : Boolean;
        attribute defaultvalue : String;
        reference ownerclass : Class oppositeOf fields;
        reference ownerSignal : Signal oppositeOf fields;
        reference fieldType : PivotVxUMLType oppositeOf typedField;
    }

class Method extends PivotVxUMLElement
    {   attribute isFinal : Boolean;
        attribute isPublic : Boolean;
        attribute isStatic : Boolean;
        reference ownerclass : Class oppositeOf method;
        reference args[*] container : Argument oppositeOf ownermethod;
        reference body[0-1] container : MethodBody oppositeOf ownedMethod;
        reference returnedtype : PivotVxUMLType;
        reference exception : Exception oppositeOf method;
    }

class MethodBody
    {   attribute expression : String;
        reference ownedMethod : Method oppositeOf body;
        reference returnedmethod : Method;
    }

class ClassBody
    {   attribute expression : String;
        reference ownedClass : Class oppositeOf body;
    }

class Exception
    {   attribute expression : String;
        reference method : Method oppositeOf exception;
    }

class Argument extends PivotVxUMLElement
    {   attribute isFinal : Boolean;
        reference ownermethod : Method oppositeOf args;
        reference argType : PivotVxUMLType oppositeOf typedargument;
    }

} ---------------------------------------------------------- Fin Package VxUML ------------------------------

package PrimitiveTypes
{
   datatype String;
   datatype Integer;
   datatype Boolean;
}
```

ANNEXE B : REGLES DE VERIFICATION SEMANTIQUE DE VXUML

```
module VxUML2Problem; -- Module Template
create OUT : Problem from VxUML : UML2;

================================================================
                Exemples de règles de vérification
================================================================

entrypoint rule CreateProblemModel()
    {
        to pm : Problem!ProblemModel    (    name <- 'ProblemModel'        )
        do {    thisModule.aModel <- pm;
                '========== Début de vérification de la cohérence du modèle VxUML ===='.output();
                ' \n'.output();
            }
    }

endpoint rule end()
    {
    do {
            ' \n'.output();
            '======= Fin de vérification de la cohérence du modèle VUML ========='.output();
        }
    }

abstract rule printNameElement
    {
        from e : UML2!Class     (not e.isInProfileDefinition)
        to p : Problem!Problem     (   description <- 'vérification de : ' + e.name  )
        do {    p.description.output();        }
    }
```

[1] Si une InstanceSpecification stéréotypée par « probe » possède un classeur, ce dernier doit être une classe stéréotypée par « probeClass ».

```
rule ProbeRule2 extends printNameElement
{   from i : VxUML2!InstanceSpecification(     not i.ProbeCondition2)
    to p : Problem!Problem
        (
        severity <-#error,
        description <- 'la sonde ' + i.name + 'doit avoir comme classifier une classe stéréotypée par « probeClass »'
        )
    do {    p.description.output();          }
}
```

```
helper context VxUML2!InstanceSpecification def : ProbeCondition2 : Boolean =
    if thisModule.inElements->includes(self) and self.hasStereotype('probe') then
        if not self.classifier.oclIsUndefined()
            then self.classifier->first().hasStereotype('probeClass')
        else true
        endif
    else  true
    endif
    ;
```

[2] Si un élément Reception est stéréotypé par « probeUse » il ne peut pas faire référence à un élément de type Signal.

```
rule ProbeRule2 extends printNameElement
{
    from r : VxUML2!Reception (    not  r.ProbeCondition2)
    to p : Problem!Problem
        (
        severity <-#error,
        description <- 'la probeUse ' + r.name + ' ne peut pas faire reference à un signal'
        )
    do {    p.description.output();          }
}

helper context VxUML2!Reception def : ProbeCondition2 : Boolean =
    if thisModule.inElements->includes(self) and self.hasStereotype('probeUse') then
        if  self.signal.oclIsUndefined()
            then false
        else true
        endif
    else  true
    endif
    ;
```

[3] Si un élément Trigger est stéréotypé par « wait » il ne peut pas faire référence à un élément de type Event.

```
rule ProbeRule3 extends printNameElement
{
    from r : VxUML2!Trigger (   not  r.ProbeCondition3)
    to p : Problem!Problem
        (
        severity <-#error,
        description <- 'le trigger ' + r.name + ' ne peut pas faire reference à un événement'
        )
    do {    p.description.output();          }
}

helper context VxUML2!Trigger def : ProbeCondition3 : Boolean =
    if thisModule.inElements->includes(self) and self.hasStereotype('wait') then
        if  self.event.oclIsUndefined()
            then false
        else true
        endif
    else  true
    endif
    ;
```

[4] Un «base» a, au moins, une relation «viewExtension», ou doit être descendant direct d'un «base» ou d'un «multiViewsClass»

```
rule BaseRule1 extends printNameElement
{
    from c : UML2!Class(    not c.BaseRuleCondition1)
    to p : Problem!Problem
        (
        severity <-#error,
        description <- 'the class ' + c.name + 'must have at least one viewExtension relationship' +' or must be
             a child of a base class'
        )
    do {    p.description.output();        }
}


helper context UML2!Class def : BaseRuleCondition1 : Boolean =
    if thisModule.inElements->includes(self)and self.hasStereotype('base') then
        if not self.getViewExtensions.oclIsUndefined()and not self.getViewExtensions->isEmpty()
            then self.getViewExtensions->select(d | d.hasStereotype('viewExtension'))->size() >=1 or
                 self.isStereokinded('base')
            else true
            endif
        else  true
    endif
    ;
```

[5] Un descendant direct de «base» est soit un «base» soit un «multiViewsClass»

```
rule BaseRule2 extends printNameElement
{
    from c : UML2!Class(    not c.BaseRuleCondition2)
    to p : Problem!Problem
        (
        severity <-#error,
        description <- 'the class ' + c.name + ' must be stereotyped by base' +' or multiViewClass'
        )
    do {    p.description.output();   }
 }


helper context UML2!Class def : BaseRuleCondition2 : Boolean =
    if thisModule.inElements->includes(self)then
        if not self.generalization.oclIsUndefined() and self.generalization->notEmpty()then
            if self.generalization->select(g | g.general.hasStereotype('base'))->size()>=1
                then self.hasStereotype('base')
            else true
            endif
        else true
        endif
    else true
    endif
    ;
```

[6] Un élément « view » ne peut hériter que de « view » ou de « abstractView »

```
rule ViewRule2 extends printNameElement
{
    from c : UML2!Class (   not c.ViewRuleCondition1)
    to  p : Problem!Problem
        (
        severity <-#error,
        description <- 'the view  ' + c.name + ' can only inherit from "view" or "abstractView" '
        )
    do {    p.description.output();        }
}

helper context UML2!Dependency def : ViewExtensionRuleCondition : Boolean =
    if thisModule.inElements->includes(self)and self.hasStereotype('viewExtension')
        then ((self.client->asSequence()->first().hasStereotype('view')
            or self.client->asSequence()->first().hasStereotype('abstractView'))
            and self.supplier->asSequence()->first().hasStereotype ('base'))
        else true
        endif
    ;
```

[7] Un descendant direct d'un « view » est soit un « view » soit un « abstractView »

```
rule ViewRule3  extends printNameElement
{
    from c : UML2!Class(    not     c.ViewRuleCondition2)
    to p : Problem!Problem
        (
        severity <-#error,
        description <- 'A direct descendant of the "view" ' + c.name + ' is either a "view" or a "abstractView" '
        )
    do {    p.description.output();   }
}

helper context UML2!Class def : ViewRuleCondition2 : Boolean =
    if thisModule.inElements->includes(self)and self.hasStereotype('view')then
        self.isStereokinded('view')or self.isStereokinded('abstractView')
    else true
    endif
    ;
```

[8] Un « view » doit être source d'une seule relation « viewExtension »,

```
rule ViewRule4 extends printNameElement
{
    from c : UML2!Class(    not c.ViewRuleCondition3)
    to p : Problem!Problem
        (
        severity <-#error,
        description <- 'the "view" '+c.name+ ' should be have a single "viewExtension" relationship, or '+
                        ' be a single child of a " view "or a single child of an " abstractView " '
        )
    do {    p.description.output();        }
}
```

```
helper context UML2!Class def : ViewRuleCondition3 : Boolean =
    if thisModule.inElements->includes(self)and self.hasStereotype('view')then
        self.clientDependency->select(r | r.hasStereotype('viewExtension'))->size() = 1
    else true
    endif
    ;
```

==========

Helpers partagés

==========

```
helper def : inElements : Set(UML2!Element) = UML2!Element->allInstancesFrom('VUML')->asSet();

helper context UML2!Element def : hasStereotype(s : String) : Boolean =
     if self.getAppliedStereotypes()->select(e | e.name = s)->notEmpty()
        then      true
        else      false
     endif
     ;

helper context UML2!Class def : isStereokinded (s : String) : Boolean =
    if not self.AllStereotypes.oclIsUndefined()then
        if self.AllStereotypes->select(e | e.name = s)->notEmpty()
            then      true
            else      false
        endif
    else false
    endif
    ;

helper context UML2!Class def : AllStereotypes : Sequence(UML2!Stereotype) =
    let stereotypes : UML2!Stereotype = self.getAppliedStereotypes() in   stereotypes->union(
        let gen : UML2!Generalization =  UML2!Generalization.allInstancesFrom('VUML')->select(g|g.specific
= self) in
            if gen.oclIsUndefined() or gen->isEmpty() then Sequence{}
            else gen->first().general.AllStereotypes
            endif
            )->flatten()
    ;

helper context UML2!Class def : allParents : Set(UML2!Element) =
    let allParents : Set(UML2!Element) = self.generalization->collect(g | g.general)->asSet() in
    allParents->union(allParents->select(x|not x.generalization.oclIsUndefined())->collect(y |y.allParents))
    ;

helper context UML2!Class def : getBase : UML2!Class =
    let vex : UML2!Dependency= self.clientDependency->select(r |r.hasStereotype('viewExtension'))-
>asSequence()in
    if not vex.oclIsUndefined() and not vex->isEmpty()
        then   vex->first().supplier
        else OclUndefined
    endif
    ;

helper def : aModel : Problem!ProblemModel = OclUndefined
    ;

helper context UML2!Class def : getViewExtensions : Sequence(UML2!Dependency)=
```

```
    let viewExt : UML2!Dependency = UML2!Dependency.allInstancesFrom('VUML')->asSequence()
        in  viewExt->iterate(p ; res : Sequence(UML2!Dependency) = Sequence{} |
        if p.hasStereotype('viewExtension') and p.supplier->first().__xmiID__ =  self.__xmiID__
            then   res->append(p)
            else res
        endif  )
    ;

helper context UML2!Element def : print() : Boolean =
    let msg : String = 'vérification de : ' + self.name in   msg.debug(msg) and     true
    ;

helper context UML2!Element def : isInProfileDefinition : Boolean =
    UML2!Element.allInstancesFrom('PRO')->collect(e | e.__xmiID__)->asSequence()-
>includes(self.__xmiID__)
    ;
```

ANNEXE C : EXEMPLE DE CODE GENERE AUTOMATIQUEMENT, LA CLASSE "VOITURE_CLIENT"

```
/*----------------------------------------------------------------------------*/
/*                    Prototype VxUML                                         */
/*                    Fichier généré automatiquement                          */
/*                    Date  : 12 - 12 - 2009                                  */
/*----------------------------------------------------------------------------*/

package application.GestionAgence;
import java.util.List;
import java.util.ArrayList;
import model.*;
import application.signalList.*;

public class Voiture_Client implements Runnable
{
    public String  couleur;
    private StateMachine stateMachine;
    public Queue queue;
    private Config config;
    private List<Signal> signalsList;
    private List<Probe> probesList;
    private ProbeControler pc;

    public String getCouleur() {        return couleur;   }
    public void setCouleur(String couleur) { this. couleur = couleur; }

    public StateMachine getStateMachine() {     return stateMachine;     }
    public void setStateMachine(StateMachine stateMachine) {this.stateMachine = stateMachine;}

    public Queue getQueue() {         return queue;}
    public void setQueue(Queue queue) {    this.queue = queue; }

    public Config getConfig() {         return config;     }
    public void setConfig(Config config) {   this.config = config;}

    public List<Signal> getSignalsList() {return signalsList;}
    public void setSignalsList(List<Signal> signalsList) {   this.signalsList = signalsList;}

    public List<Probe> getProbesList() {     return probesList;}
    public void setProbesList(List<Probe> probesList) {  this.probesList = probesList;}

    public ProbeControler getPc() {   return pc;}
    public void setPc(ProbeControler pc) {  this.pc = pc;}
```

```
public List<Vertex> getIntialStates()
    {
        List<Vertex> a= new ArrayList<Vertex>(1);
        for(int i=0; i<this.stateMachine.regionCount();i++)
            {
                a.add(this.stateMachine.getRegion(i).getVertex(0));
            }
        return a;
    }


//-------------------ajout des guardes des transitions----------------------

public static IConstraint guard1 = new Constraint()
    {
        public boolean eval(Object obj)
            {
                Voiture_Client obj1= (Voiture_Client)obj;
                return true;
            }
        public String affiche()
            {
                return "";
            }
    };


public static IConstraint c = new Constraint()
    {
        public boolean eval(Object obj)
            {
                Voiture_Client obj1= (Voiture_Client)obj;
                return true;
            }
        public String affiche()
            {
                return "";
            }
    };

public static IConstraint guard2 = new Constraint()
    {
        public boolean eval(Object obj)
            {
                Voiture_Client obj1= (Voiture_Client)obj;
                return true;
            }
        public String affiche()
            {
                return "";
            }
    };

//-------------------ajout des effects des transitions----------------------

public static IBehavior  effet1  = new Behavior()
    {
```

```
        public void exec(Object obj)
            {
                Voiture_Client obj1 = (Voiture_Client)obj;
            }
        public String affiche()
            {
                return " ";
            }
    };

public static IBehavior  e  = new Behavior()
    {
        public void exec(Object obj)
            {
                Voiture_Client obj1 = (Voiture_Client)obj;
                //pas encore traité
            }
        public String affiche()
            {
                return " ";
            }
    };

public static IBehavior  effet2  = new Behavior()
    {
        public void exec(Object obj)
            {
                Voiture_Client obj1 = (Voiture_Client)obj;
                //pas encore traité
            }
        public String affiche()
            {
                return " ";
            }
    };

//-------------------Creation de la machine à états ------------------------

public void createSM()
    {
    Signal sig=new Signal();Transition tr =new Transition();

    Region     reg_1_client = new Region("reg_1_client",this.stateMachine);
        /*  Ajout Etat    */ reg_1_client.addVertex( new StateV("state1_r1", reg_1_client) );
        /*  Ajout Etat    */ reg_1_client.addVertex( new StateV("state3_r1", reg_1_client) );
        /*  Ajout Etat    */ reg_1_client.addVertex( new StateV("state2_r1", reg_1_client) );

        /*Ajout Transiton */ tr = new Transition("tr1",
                                    reg_1_client.getVertex(reg_1_client.trouverRangEtat("state1_r1")),
                                    reg_1_client.getVertex(reg_1_client.trouverRangEtat("state1_r1") ),
                                    reg_1_client   );
                    tr.setConstraint(guard1);
                    tr.setEffect(effet1);
                    sig = new S1();
                    signalsList.add(sig);
                    tr.setTrigger(new Trigger(sig));
                    reg_1_client.addTransition(tr);
```

```
      /*Ajout Transiton */ tr = new Transition("tr2",
                                    reg_1_client.getVertex(reg_1_client.trouverRangEtat("state2_r1")),
                                    reg_1_client.getVertex(reg_1_client.trouverRangEtat("state2_r1")),
                                    reg_1_client);
                        tr.setConstraint(c);
                        tr.setEffect(e);
                        tr.setTrigger(new Trigger());
                        reg_1_client.addTransition(tr);
    this.stateMachine.addRegion(reg_1_client);

    Region     reg_2_client = new Region("reg_2_client",this.stateMachine);
        /*  Ajout Etat    */ reg_2_client.addVertex( new StateV("state1_r2", reg_2_client) );
        /*  Ajout Etat    */ reg_2_client.addVertex( new StateV("state2_r2", reg_2_client) );

      /*Ajout Transiton */ tr = new Transition("tr1",
                                    reg_2_client.getVertex(reg_2_client.trouverRangEtat("state1_r2") ),
                                    reg_2_client.getVertex(reg_2_client.trouverRangEtat("state1_r2") ),
                                    reg_2_client);
                        tr.setConstraint(guard2);
                        tr.setEffect(effet2);
                        sig = new S2();
                        signalsList.add(sig);
                        tr.setTrigger(new Trigger(sig));
                        reg_2_client.addTransition(tr);
    this.stateMachine.addRegion(reg_2_client);
    }

} //---  fin de la classe  "Voiture_Client"
```